

**SAMS**

畅销全球的  
**经典C++教程**

# 21天学通 C++ (第6版)

Jesse Liberty  
[美] Siddhartha Rao 著  
Bradley Jones  
袁国忠 陈秋萍 译

全美销量超过25万册



人民邮电出版社  
POSTS & TELECOM PRESS

# 21天学通 C++ (第6版)

Jesse Liberty  
[美] Siddhartha Rao 著  
Bradley Jones  
袁国忠 陈秋萍 译





## 图书在版编目 (C I P) 数据

21天学通C++: 第6版 / (美) 利伯蒂 (Liberty, J.),  
(美) 拉奥 (Rao, S.), (美) 琼斯 (Jones, B.) 著;  
袁国忠, 陈秋萍译. —北京: 人民邮电出版社, 2009. 8  
ISBN 978-7-115-20793-7

I. 2… II. ①利…②拉…③琼…④袁…⑤陈… III. C语  
言—程序设计 IV. TP312

中国版本图书馆CIP数据核字 (2009) 第087275号

### 版 权 声 明

Jesse Liberty, Siddhartha Rao, Bradley Jones: Sams Teach Yourself C++ in One Hour a Day

ISBN: 978-0-672-32941-8

Copyright © 2009 by Sams Publishing

Authorized translation from the English language edition published by Sams.

All rights reserved.

本书中文简体字版由美国 Sams 出版公司授权人民邮电出版社出版。未经出版者书面许可, 对本书任何部分不得以任何方式复制或抄袭。

版权所有, 侵权必究。

## 21 天学通 C++ (第 6 版)

◆ 著 [美] Jesse Liberty Siddhartha Rao  
Bradley Jones

译 袁国忠 陈秋萍

责任编辑 李 际

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号

邮编 100061 电子函件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京鸿佳印刷厂印刷

◆ 开本: 787×1092 1/16

印张: 31.5

字数: 931 千字

2009 年 8 月第 1 版

印数: 1—4 000 册

2009 年 8 月北京第 1 次印刷

著作权合同登记号 图字: 01-2008-3320 号

ISBN 978-7-115-20793-7/TP

定价: 55.00 元

读者服务热线: (010)67132705 印装质量热线: (010)67129223

反盗版热线: (010)67171154

# 内容提要

本书通过大量短小精悍的程序详细而全面地阐述了 C++ 的基本概念和技术，包括管理输入/输出、循环和数组、面向对象编程、模板、使用标准模板库以及创建 C++ 应用程序等。这些内容被组织成结构合理、联系紧密的章节，每章都可在 1 小时内阅读完毕；每章都提供了示例程序清单，并辅以示例输出和代码分析，以阐述该章介绍的主题。为加深读者对所学内容的理解，每章末尾都提供了常见问题及其答案以及练习和测验。读者可对照附录 D 提供的测验和练习答案，了解自己对所学内容的掌握程度。

本书是针对 C++ 初学者编写的，不要求读者有 C 语言方面的背景知识，可作为高等院校教授 C++ 课程的教材，也可供初学者自学 C++ 时使用。

资源分享  
PDG

# 作者简介

Siddhartha Rao 是一位微软 Visual C++ MVP，还是最活跃的 Internet 开发社区之一 CodeGuru 的主持人。他是 Windows 编程领域的专家，在架构设计以及使用 C++ 和其他现代编程语言开发驱动程序和应用程序方面拥有丰富的经验。当前，他为德国的一家软件巨人工作，致力于软件管理和软件开发最佳实践。鉴于在 3 个国家居住和生活过，他认为自己和家人得了旅行狂热症。Siddhartha 能说多种语言，闲暇期间他喜欢在全球各地旅行和摄影。

Jesse Liberty 编著了大量有关软件开发的图书，其中包括 C++ 和 .NET 方面的畅销书。他是 Liberty Associates 公司的总裁，该公司致力于为客户提供编程、咨询和培训方面的服务。

Bradley Jones 是一位微软 Visual C++ MVP，他身兼网站管理员、经理、编码大师、执行编辑等职，其主要精力放在众多软件开发网站和频道上，其中包括 Developer.com、CodeGuru.com、DevX、VBForums、Gamelan 以及 Jupitermedia 的其他网站。

资源分享网  
PDG

# 前言

本书旨在帮助读者学习如何使用 C++ 进行编程。就像人需要慢慢学会走路一样，学习 C++ 编程也需要循序渐进，因此本书每章包含的内容都可以在 1 小时内阅读完毕。本书通过实际使用 C++，帮助读者快速掌握编写实用的 C++ 应用程序涉及的最重要的概念。

通过每天学习 1 小时，读者将逐步掌握管理输入/输出、循环和数组、面向对象编程、模板、使用标准模板库以及创建 C++ 应用程序等基本知识，所有这些内容都组织成结构合理、易于理解的章节。每章都提供示例程序清单，并辅以示例输出和代码分析以演示该章介绍的主题。

为加深读者对所学内容的理解，每章末尾都提供了常见问题及其答案以及练习和测验。读者可对照附录 D 提供的测验和练习答案，了解自己对所学内容的掌握程度。

## 针对的读者

通过阅读本书来学习 C++ 时，读者不需要有任何编程经验。本书从入门开始，既介绍 C++ 语言，又讨论使用 C++ 进行编程涉及的概念。本书提供了大量语法实例和详细的代码分析，它们是引导读者完成 C++ 编程之旅的优秀向导。无论读者是刚开始学习编程还是已经有一些编程经验，书中精心安排的内容都将让您的 C++ 学习过程变得既快速又轻松。

## 本书内容

本书适合初学者阅读，也可供有一定经验的 C++ 程序员从实用角度更深入了解 C++ 时参考。本书包含 5 部分：

第一部分简要地介绍了 C++ 及其语法，这对于需要学习 C++ 编程基本知识的读者极具参考价值。

第二部分简要地介绍了 C++ 的面向对象编程功能，这些功能让 C++ 不同于其前身 C 语言。这部分将为实际使用 C++ 及其标准模板库打下坚实的基础。

第三部分深入探讨了如何使用 C++ 编写实用的应用程序，通过使用符合标准的现成结构，可极大地改善应用程序的质量。

第四部分简要地介绍了诸如排序等 STL 算法以及其他 STL 结构，它们有助于改善应用程序的效率和可靠性。

第五部分详细讨论了 C++ 的一些高级功能。虽然并非编写每个应用程序都涉及这些概念，但了解它们有助于分析错误以及编写出质量更高的代码。

## 本书约定

在本书中，遍布如下提供更多信息的元素：

### 提示

提供使读者进行 C++ 编程时更高效、更有效的信息。



**注意** 提供与读者阅读的内容相关的信息。

**FAQ**

对 C++ 语言的用法进行了深入剖析，澄清一些容易混淆的问题。

**警告** 提醒读者注意在特定情况下可能出现的问题或副作用。

应该	不应该
提供当前章介绍的基本原理的摘要。	提供一些有用的信息。

**示例代码**

本书正文及附录 D 中的示例代码可从人民邮电出版社网站下载，网址为 <http://www.ptpress.com.cn>。



# 目 录

## 第一部分 基础知识

第 1 章 绪论	2	2.5.1 使用函数	19
1.1 C++简史	2	2.5.2 方法和函数	21
1.1.1 解释器和编译器	3	2.6 总结	21
1.1.2 不断变化的需求和平台	3	2.7 问与答	21
1.1.3 过程化编程、结构化编程和面向 对象编程	4	2.8 作业	21
1.1.4 面向对象编程 (OOP)	4	2.8.1 测验	21
1.1.5 C++和面向对象编程	4	2.8.2 练习	21
1.2 C++的发展历程	5	第 3 章 使用变量和常量	23
1.3 应该先学习 C 语言吗	5	3.1 什么是变量	23
1.4 微软的 C++托管扩展	6	3.1.1 将数据存储在内存中	23
1.5 ANSI 标准	6	3.1.2 预留内存	24
1.6 编程准备	6	3.1.3 整型变量的大小	24
1.7 开发环境	7	3.1.4 基本变量类型	24
1.8 创建程序的步骤	7	3.2 定义变量	25
1.8.1 用编译器生成对象文件	7	3.2.1 区分大小写	26
1.8.2 用链接器生成可执行文件	7	3.2.2 命名规则	26
1.9 程序开发周期	8	3.2.3 关键字	27
1.10 HELLO.cpp: 第一个 C++程序	9	3.3 确定变量类型占用的内存量	27
1.11 编译器初步	10	3.4 一次创建多个变量	28
1.12 编译错误	11	3.5 给变量赋值	28
1.13 总结	11	3.6 使用 typedef 创建别名	29
1.14 问与答	11	3.7 何时使用 short 和 long	30
1.15 作业	12	3.7.1 unsigned 整型变量的回绕	31
1.15.1 测验	12	3.7.2 signed 整型变量的回绕	31
1.15.2 练习	12	3.8 使用字符	32
第 2 章 C++程序的组成部分	13	3.8.1 字符和数字	32
2.1 一个简单程序	13	3.8.2 特殊打印字符	33
2.2 cout 简介	14	3.9 常量	34
2.3 使用标准名称空间	16	3.9.1 字面常量	34
2.4 对程序进行注释	17	3.9.2 符号常量	34
2.4.1 注释的类型	17	3.10 枚举常量	35
2.4.2 使用注释	18	3.11 总结	36
2.4.3 有关注释的警告	18	3.12 问与答	37
2.5 函数	18	3.13 作业	37
		3.13.1 测验	37
		3.13.2 练习	38

第 4 章 管理数组和字符串	39	5.15 条件运算符 (三目运算符)	70
4.1 什么是数组	39	5.16 总结	71
4.1.1 访问数组元素	39	5.17 问与答	71
4.1.2 在数组末尾后写入数据	40	5.18 作业	71
4.1.3 护栏柱错误	42	5.18.1 测验	71
4.1.4 初始化数组	42	5.18.2 练习	72
4.1.5 声明数组	43	第 6 章 使用函数组织代码	73
4.2 使用多维数组	44	6.1 什么是函数	73
4.2.1 声明多维数组	44	6.2 返回值、参数和实参	74
4.2.2 初始化多维数组	44	6.3 声明和定义函数	74
4.3 字符数组和字符串	46	6.3.1 函数原型	74
4.4 使用方法 strcpy() 和 strncpy()	48	6.3.2 定义函数	75
4.5 string 类	49	6.4 函数的执行	76
4.6 总结	50	6.5 确定变量的作用域	77
4.7 问与答	51	6.5.1 局部变量	77
4.8 作业	51	6.5.2 作用域为语句块的局部变量	78
4.8.1 测验	51	6.6 参数是局部变量	79
4.8.2 练习	51	6.6.1 全局变量	80
第 5 章 使用表达式、语句和运算符	53	6.6.2 有关全局变量的注意事项	81
5.1 语句简介	53	6.7 创建函数语句时的考虑因素	81
5.1.1 使用空白	53	6.8 再谈函数实参	81
5.1.2 语句块和复合语句	54	6.9 再谈返回值	82
5.2 表达式	54	6.10 默认参数	83
5.3 使用运算符	55	6.11 重载函数	85
5.3.1 赋值运算符	55	6.12 函数特有的主题	87
5.3.2 数学运算符	55	6.12.1 内联函数	87
5.3.3 整数除法和求模	56	6.12.2 递归	89
5.4 结合使用赋值运算符与数学运算符	57	6.13 函数的工作原理	92
5.5 递增和递减	57	6.13.1 抽象层次	92
5.6 理解运算符优先级	59	6.13.2 划分 RAM	92
5.7 括号的嵌套	59	6.13.3 堆栈和函数	93
5.8 真值的本质	60	6.14 总结	94
5.9 if 语句	61	6.15 问与答	94
5.9.1 缩进风格	63	6.16 作业	95
5.9.2 else 语句	63	6.16.1 测验	95
5.9.3 高级 if 语句	65	6.16.2 练习	95
5.10 在嵌套 if 语句中使用大括号	66	第 7 章 控制程序流程	97
5.11 使用逻辑运算符	68	7.1 循环	97
5.11.1 逻辑 AND 运算符	68	7.1.1 循环的鼻祖: goto	97
5.11.2 逻辑 OR 运算符	68	7.1.2 为何避免使用 goto 语句	98
5.11.3 逻辑 NOT 运算符	68	7.2 使用 while 循环	98
5.12 简化求值	68	7.2.1 更复杂的 while 语句	99
5.13 关系运算符的优先级	69	7.2.2 continue 和 break 简介	100
5.14 再谈真和假	69	7.2.3 while(true) 循环	102

7.3 实现 do...while 循环.....	103	8.3.2 使用关键字 delete 归还内存.....	128
7.4 使用 do...while.....	103	8.4 再谈内存泄漏.....	130
7.5 for 循环.....	105	8.5 在自由存储区上创建对象.....	130
7.5.1 高级 for 循环.....	106	8.6 删除自由存储区中的对象.....	130
7.5.2 空 for 循环.....	108	8.7 迷途指针.....	131
7.5.3 循环嵌套.....	109	8.8 使用 const 指针.....	133
7.5.4 for 循环中声明的变量的作用域.....	110	8.9 总结.....	134
7.6 循环小结.....	111	8.10 问与答.....	134
7.7 使用 switch 语句控制程序流程.....	112	8.11 作业.....	134
7.8 总结.....	116	8.11.1 测验.....	134
7.9 问与答.....	117	8.11.2 练习.....	135
7.10 作业.....	117	第 9 章 使用引用.....	136
7.10.1 测验.....	117	9.1 什么是引用.....	136
7.10.2 练习.....	117	9.2 将地址运算符用于引用.....	137
第 8 章 阐述指针.....	119	9.3 空指针和空引用.....	139
8.1 什么是指针.....	119	9.4 按引用传递函数参数.....	139
8.1.1 内存简介.....	119	9.4.1 使用指针让 swap() 管用.....	140
8.1.2 获取变量的内存地址.....	120	9.4.2 使用引用来实现 swap().....	141
8.1.3 将变量的地址存储到指针中.....	120	9.5 返回多个值.....	142
8.1.4 指针名.....	121	9.6 按引用传递以提高效率.....	145
8.1.5 获取指针指向的变量的值.....	121	9.6.1 传递 const 指针.....	147
8.1.6 使用间接运算符解除引用.....	122	9.6.2 用引用代替指针.....	148
8.1.7 指针、地址和变量.....	122	9.7 何时使用引用和指针.....	150
8.1.8 使用指针来操纵数据.....	123	9.8 混合使用引用和指针.....	150
8.1.9 查看地址.....	124	9.9 返回指向不在作用域中的对象的引用.....	151
8.1.10 指针和数组名.....	125	9.10 总结.....	153
8.1.11 数组指针和指针数组.....	126	9.11 问与答.....	153
8.2 为什么使用指针.....	127	9.12 作业.....	153
8.3 栈和自由存储区(堆).....	127	9.12.1 测验.....	154
8.3.1 使用关键字 new 分配内存.....	128	9.12.2 练习.....	154

## 第二部分 面向对象编程和 C++ 基础

第 10 章 类和对象.....	156	10.6 实现类方法.....	163
10.1 C++ 是面向对象的吗.....	156	10.7 添加构造函数和析构函数.....	165
10.2 创建新类型.....	157	10.7.1 默认构造函数和析构函数.....	166
10.3 类和成员简介.....	157	10.7.2 使用默认构造函数.....	166
10.3.1 声明类.....	158	10.8 const 成员函数.....	168
10.3.2 有关命名规则的说明.....	158	10.9 将类声明和方法定义放在什么地方.....	169
10.3.3 定义对象.....	159	10.10 内联实现.....	169
10.3.4 类与对象.....	159	10.11 将其他类用作成员数据的类.....	171
10.4 访问类成员.....	159	10.12 探索结构.....	174
10.4.1 给对象而不是类赋值.....	159	10.13 总结.....	174
10.4.2 类不能有没有声明的功能.....	159	10.14 问与答.....	174
10.5 私有和公有.....	160	10.15 作业.....	175



10.15.1 测验	175	12.3.3 复杂的抽象层次结构	224
10.15.2 练习	176	12.3.4 哪些类是抽象的	226
第 11 章 实现继承	177	12.4 总结	226
11.1 什么是继承	177	12.5 问与答	227
11.1.1 继承和派生	177	12.6 作业	227
11.1.2 动物世界	178	12.6.1 测验	227
11.1.3 派生的语法	178	12.6.2 练习	228
11.2 私有和保护	180	第 13 章 运算符类型与运算符重载	229
11.3 构造函数和析构函数的继承性	181	13.1 C++中的运算符	229
11.4 覆盖基类函数	186	13.2 单目运算符	229
11.4.1 隐藏基类的方法	187	13.2.1 单目运算符的类型	230
11.4.2 调用基类方法	189	13.2.2 单目递增与单目递减运算符	230
11.5 虚方法	190	13.2.3 解除引用运算符*与成员选择 运算符->的编程	232
11.5.1 虚函数的工作原理	193	13.2.4 转换运算符的编程	234
11.5.2 通过基类指针访问派生类的 方法	193	13.3 双目运算符	235
11.5.3 切除	194	13.3.1 双目运算符的类型	235
11.5.4 创建虚析构函数	195	13.3.2 双目加与双目减运算符的编程	236
11.5.5 虚复制构造函数	195	13.3.3 运算符+=与-=的编程	237
11.5.6 使用虚方法的代价	198	13.3.4 重载比较运算符	238
11.6 私有继承	198	13.3.5 重载运算符<、>、<=和>=	241
11.6.1 使用私有继承	198	13.3.6 下标运算符	243
11.6.2 私有继承和聚合(组合)	199	13.4 operator()函数	244
11.7 总结	200	13.5 不能重新定义的运算符	245
11.8 问与答	201	13.6 总结	245
11.9 作业	201	13.7 问与答	245
11.9.1 测验	201	13.8 作业	246
11.9.2 练习	202	13.8.1 测验	246
第 12 章 多态	203	13.8.2 练习	246
12.1 单继承存在的问题	203	第 14 章 类型转换运算符	247
12.1.1 提升	205	14.1 什么是类型转换	247
12.1.2 向下转换	205	14.2 为何需要类型转换	247
12.1.3 将对象添加到链表中	207	14.3 为何有些 C++程序员不喜欢 C 风格 类型转换	248
12.2 多重继承	207	14.4 C++类型转换运算符	248
12.2.1 多重继承对象的组成部分	209	14.4.1 使用 static_cast	248
12.2.2 多重继承对象中的构造函数	210	14.4.2 使用 dynamic_cast 和运行阶段 类型识别	249
12.2.3 避免歧义	212	14.4.3 使用 reinterpret_cast	250
12.2.4 从共同基类继承	212	14.4.4 使用 const_cast	251
12.2.5 虚继承	215	14.5 C++类型转换运算符存在的问题	252
12.2.6 多重继承存在的问题	217	14.6 总结	252
12.2.7 混合(功能)类	217	14.7 问与答	252
12.3 抽象数据类型	218	14.8 作业	253
12.3.1 纯虚函数	220		
12.3.2 实现纯虚函数	221		

第 15 章 宏和模板简介 .....	254
15.1 预处理器与编译器 .....	254
15.2 预处理器指令#define .....	254
15.3 宏函数 .....	255
15.3.1 为什么要使用括号 .....	255
15.3.2 宏与类型安全问题 .....	256
15.3.3 宏与函数及模板之比较 .....	256
15.3.4 内联函数 .....	256
15.4 模板简介 .....	258
15.4.1 模板声明语法 .....	258
15.4.2 各种类型的模板声明 .....	258
15.4.3 模板类 .....	259
15.4.4 模板的实例化和具体化 .....	259
15.4.5 模板与类型安全 .....	259
15.4.6 使用多个参数声明模板 .....	259
15.4.7 使用默认参数来声明模板 .....	260
15.4.8 一个模板示例 .....	260
15.4.9 在实际 C++ 编程中使用模板 .....	261
15.5 总结 .....	262
15.6 问与答 .....	262
15.7 作业 .....	263
15.7.1 测验 .....	263
15.7.2 练习 .....	263

### 第三部分 学习标准模板库 (STL)

第 16 章 标准模板库简介 .....	266
16.1 STL 容器 .....	266
16.1.1 顺序容器 .....	266
16.1.2 关联容器 .....	266
16.1.3 选择正确的容器 .....	267
16.2 STL 迭代器 .....	267
16.3 STL 算法 .....	268
16.4 使用迭代器在容器和算法之间交互 .....	268
16.5 总结 .....	270
16.6 问与答 .....	270
16.7 作业 .....	270
第 17 章 STL string 类 .....	271
17.1 为何需要字符串操作类 .....	271
17.2 使用 STL string 类 .....	272
17.2.1 实例化 STL string 及复制 .....	272
17.2.2 访问 string 及其内容 .....	273
17.2.3 字符串连接 .....	274
17.2.4 在 string 中查找字符或子字符串 .....	275
17.2.5 截短 STL string .....	276
17.2.6 字符串反转 .....	278
17.2.7 字符串的大小写转换 .....	278
17.3 基于模板的 STL string 实现 .....	279
17.4 总结 .....	279
17.5 问与答 .....	279
17.6 作业 .....	280
17.6.1 测验 .....	280
17.6.2 练习 .....	280
第 18 章 STL 动态数组类 .....	281
18.1 std::vector 的特点 .....	281
18.2 典型的 vector 操作 .....	281
18.2.1 实例化 vector .....	281
18.2.2 在 vector 中插入元素 .....	282
18.2.3 访问 vector 中的元素 .....	285
18.2.4 删除 vector 中的元素 .....	286
18.3 理解 size() 和 capacity() .....	287
18.4 STL deque 类 .....	288
18.5 总结 .....	290
18.6 问与答 .....	290
18.7 作业 .....	290
18.7.1 测验 .....	291
18.7.2 练习 .....	291
第 19 章 STL list .....	292
19.1 std::list 的特点 .....	292
19.2 基本的 list 操作 .....	292
19.2.1 实例化 std::list 对象 .....	292
19.2.2 在 list 开头插入元素 .....	293
19.2.3 在 list 末尾插入元素 .....	293
19.2.4 在 list 中间插入元素 .....	294
19.2.5 删除 list 中的元素 .....	296
19.3 对 list 中元素进行反转和排序 .....	297
19.3.1 反转元素的排列顺序 .....	297
19.3.2 元素排序 .....	298
19.4 总结 .....	305
19.5 问与答 .....	305
19.6 作业 .....	305
19.6.1 测验 .....	305
19.6.2 练习 .....	305

第 20 章 STL set 与 multiset	306
20.1 简介	306
20.2 STL set 和 multiset 的基本操作	306
20.2.1 实例化 std::set 对象	306
20.2.2 在 STL set 或 multiset 中插入元素	307
20.2.3 在 STL set 或 multiset 中查找元素	308
20.2.4 删除 STL set 或 multiset 中的元素	309
20.3 使用 STL set 和 multiset 的优缺点	315
20.4 总结	316
20.5 问与答	316
20.6 作业	316
20.6.1 测验	316
20.6.2 练习	316

第 21 章 STL map 和 multimap	317
21.1 简介	317
21.2 STL map 和 multimap 的基本操作	317
21.2.1 实例化 std::map 对象	317
21.2.2 在 STL map 或 multimap 中插入元素	318
21.2.3 在 STL map 或 multimap 中查找元素	320
21.2.4 删除 STL map 或 multimap 中的元素	321
21.3 提供自定义的排序谓词	323
21.4 总结	325
21.5 问与答	325
21.6 作业	326
21.6.1 测验	326
21.6.2 练习	326

## 第四部分 再谈 STL

第 22 章 理解函数对象	328
22.1 函数对象与谓词的概念	328
22.2 函数对象的典型用途	328
22.2.1 一元函数	328
22.2.2 一元谓词	331
22.2.3 二元函数	332
22.2.4 二元谓词	334
22.3 总结	336
22.4 问与答	336
22.5 作业	336
22.5.1 测验	336
22.5.2 练习	336
第 23 章 STL 算法	337
23.1 什么是 STL 算法	337
23.2 STL 算法的分类	337
23.2.1 非变序算法	337
23.2.2 变序算法	338
23.3 STL 算法的应用	339
23.3.1 计算元素个数与查找元素	339
23.3.2 在集合中搜索元素或序列	340
23.3.3 将容器中的元素初始化为指定值	342
23.3.4 用 for_each 处理范围内的元素	344
23.3.5 使用 std::transform 对范围进行变换	345

23.3.6 复制和删除操作	347
23.3.7 替换值以及替换满足给定条件的元素	349
23.3.8 排序、在有序集合中搜索以及删除重复元素	350
23.3.9 将范围分区	351
23.3.10 在有序集合中插入元素	353
23.4 总结	354
23.5 问与答	354
23.6 作业	355
23.6.1 测验	355
23.6.2 练习	355
第 24 章 自适应容器：栈和队列	356
24.1 栈和队列的行为特征	356
24.1.1 栈	356
24.1.2 队列	356
24.2 使用 STL stack 类	356
24.2.1 实例化 stack	357
24.2.2 stack 的成员函数	357
24.3 使用 STL queue 类	359
24.3.1 实例化 queue	359
24.3.2 queue 的成员函数	359
24.4 使用 STL 优先级队列	361
24.4.1 实例化 priority_queue 类	361
24.4.2 priority_queue 的成员函数	362

24.5 总结	364	25.2.2 std::bitset 的成员方法	366
24.6 问与答	364	25.3 vector<bool>	368
24.7 作业	364	25.3.1 实例化 vector<bool>	368
24.7.1 测验	364	25.3.2 使用 vector<bool>	369
24.7.2 练习	364	25.4 总结	370
第 25 章 使用 STL 位标志	365	25.5 问与答	370
25.1 bitset 类	365	25.6 作业	370
25.2 使用 std::bitset 及其成员	366	25.6.1 测验	370
25.2.1 std::bitset 的运算符	366	25.6.2 练习	370

## 第五部分 高级 C++ 概念

第 26 章 理解智能指针	372	27.6.1 单字符输入	386
26.1 什么是智能指针	372	27.6.2 从标准输入读取字符串	388
26.1.1 使用常规 (原始) 指针有何问题	372	27.6.3 使用 cin.ignore()	390
26.1.2 智能指针有何帮助	372	27.6.4 查看和插入字符: peek() 和 putback()	391
26.2 智能指针是如何实现的	373	27.7 使用 cout 进行输出	391
26.3 智能指针类型	374	27.7.1 刷新输出	391
26.3.1 深度复制	374	27.7.2 执行输出的函数	392
26.3.2 写时复制机制	375	27.7.3 控制符、标记和格式化指令	393
26.3.3 引用计数智能指针	375	27.8 流和 printf() 函数之比较	396
26.3.4 引用链接智能指针	376	27.9 文件输入和输出	398
26.3.5 破坏性复制	376	27.9.1 使用 ofstream	398
26.4 使用 std::auto_ptr	377	27.9.2 条件状态	398
26.5 流行的智能指针库	378	27.9.3 打开文件进行输入和输出	398
26.6 总结	378	27.9.4 修改 ofstream 打开文件时的默认行为	400
26.7 问与答	379	27.10 二进制文件和文本文件	401
26.8 作业	379	27.11 命令行处理	403
26.8.1 测试	379	27.12 总结	405
26.8.2 练习	379	27.13 问与答	405
第 27 章 处理流	380	27.14 作业	406
27.1 流概述	380	27.14.1 测验	406
27.1.1 数据流的封装	380	27.14.2 练习	406
27.1.2 理解缓冲技术	381	第 28 章 处理异常	407
27.2 流和缓冲区	382	28.1 程序中的各种错误	407
27.3 标准 I/O 对象	382	28.2 异常的基本思想	408
27.4 重定向标准流	382	28.2.1 异常处理的组成部分	409
27.5 使用 cin 进行输入	382	28.2.2 手工引发异常	411
27.5.1 输入字符串	384	28.2.3 创建异常类	412
27.5.2 字符串的问题	384	28.3 使用 try 块和 catch 块	414
27.5.3 >> 的返回值	386	28.4 捕获异常的工作原理	415
27.6 cin 的其他成员函数	386	28.4.1 使用多条 catch 语句	415



28.4.2 异常层次结构	417
28.5 异常中的数据及给异常对象命名	419
28.6 异常和模板	424
28.7 没有错误的异常	426
28.8 bug 和调试	426
28.8.1 断点	427
28.8.2 监视点	427
28.8.3 查看内存	427
28.8.4 查看汇编代码	427
28.9 总结	427
28.10 问与答	427
28.11 作业	428
28.11.1 测验	428
28.11.2 练习	428
<b>第 29 章 杂项内容</b>	<b>430</b>
29.1 预处理器和编译器	430
29.2 预编译器指令#define	430
29.2.1 使用#define 定义常量	431
29.2.2 将#define 用于检测	431
29.2.3 预编译器命令#else	431
29.3 包含和防范多重包含	432
29.4 字符串操纵	433
29.4.1 字符串化	433
29.4.2 拼接	433
29.5 预定义的宏	433
29.6 assert() 宏	434
29.6.1 使用 assert() 进行调试	435
29.6.2 assert() 与异常之比较	435
29.6.3 副作用	435
29.6.4 类的不变量	436
29.6.5 打印中间值	439
29.7 位运算	440
29.7.1 “与”运算符	441
29.7.2 “或”运算符	441
29.7.3 “异或”运算符	441
29.7.4 “求反”运算符	441
29.7.5 设置位	441
29.7.6 清除位	441

29.7.7 反转位	442
29.7.8 位字段	442
29.8 编程风格	444
29.8.1 缩进	444
29.8.2 大括号	444
29.8.3 长代码行和函数长度	445
29.8.4 格式化 switch 语句	445
29.8.5 程序文本	445
29.8.6 标识符命名	446
29.8.7 名称的拼写和大写	446
29.8.8 注释	446
29.8.9 设置访问权限	447
29.8.10 类定义	447
29.8.11 包含文件	447
29.8.12 使用 assert()	447
29.8.13 使用 const	447
29.9 C++ 开发工作的下一步	447
29.9.1 从何处获得帮助和建议	448
29.9.2 相关的 C++ 主题: 托管 C++、 C# 和微软的 .NET	448
29.10 总结	448
29.11 问与答	449
29.12 作业	450
29.12.1 测验	450
29.12.2 练习	450

<b>附录 A 二进制和十六进制</b>	<b>451</b>
----------------------	------------

A.1 其他进制	451
A.2 不同进制之间的转换	452
A.2.1 二进制	452
A.2.2 为什么使用二进制	453
A.2.3 位、字节和半字节	453
A.2.4 什么是 KB	453
A.2.5 二进制数	454
A.3 十六进制	454

<b>附录 B C++ 关键字</b>	<b>457</b>
---------------------	------------

<b>附录 C 运算符优先级</b>	<b>458</b>
--------------------	------------

<b>附录 D 答案</b>	<b>459</b>
----------------	------------



# 第一部分

## 基础知识

第 1 章 绪论

第 2 章 C++程序的组成部分

第 3 章 使用变量和常量

第 4 章 管理数组和字符串

第 5 章 使用表达式、语句和运算符

第 6 章 使用函数组织代码

第 7 章 控制程序流程

第 8 章 阐述指针

第 9 章 使用引用

# 第1章

## 绪论

欢迎使用本书！通过阅读本章，您将迈出成为高级 C++ 程序员的第一步。

在本章中，您将学习：

- 为何 C++ 是软件开发的标准
- 开发 C++ 程序的步骤
- 输入、编译和链接第一个 C++ 程序

### 1.1 C++ 简史

自第一代电子计算机诞生后，计算机语言经历了翻天覆地的变化。起初，程序员们使用最原始的计算机指令，即机器语言，这些指令是由 0 和 1 组成的字符串。很快，人们就发明了汇编语言，将机器指令映射为人们可以阅读和易于处理的助记符，如 ADD 和 MOV。

然而，随着编写的软件应用程序执行的任务日益复杂（如计算弹道），程序员意识到需要一种能够执行相对复杂的数学指令的语言，这些数学指令可转换为众多的汇编代码（机器语言指令）。FORTRAN 应运而生，它是编程领域中第一种针对数值和科学计算进行了优化的高级编程语言，支持子程序、函数和循环等。随后出现了更高级的语言，如 BASIC 和 COBOL，它们让程序员能够使用类似于单词或句子的源代码（如 Let I=100）进行编程。

C 语言对 B 语言做了革命性改进，而 B 语言是 BCPL (Basic Combined Programming Language) 语言的改进版本。虽然发明 C 语言旨在帮助程序员使用当时新出现的硬件功能，但它得以流行应主要归功于其可移植性和速度。C 语言是一种过程化语言，但随着计算机语言进入面向对象时代，Bjarne Stroustrup 于 1981 年发明了 C++，它是发展最快、使用最广泛的编程语言之一。除新增了诸如运算符重载和内联函数等功能外，C++ 还实现了诸如继承（支持多继承）、封装、抽象和多态等面向对象概念。C++ 还实现并不断改进了模板（泛型类或函数）概念，而诸如 Java 和 C# 等较新的语言直到最近才支持这种概念。

在 C++ 之后，Java 给编程领域带来了又一次革命。它得以流行的主要原因是 Java 应用程序可在多种流行的平台中运行；另一个原因是其简单性，它不支持众多让 C++ 功能强大的功能。除不支持指针外，Java 还负责为用户管理内存和执行垃圾收集。在 Java 之后，C# 是最先开发的基于框架（微软 .NET 框架）的语言之一。C# 借鉴了 Java 和 C++ 的设计思想和语法，但在有些方面与这两种语言都不同。.NET 框架支持管理版 C++（称为托管 C++），它向 C++ 程序员提供了 .NET 框架的优点（如自动管理内存和收集垃圾），且执行速度比其他基于框架的语言（如 C#）快。

当前，很多应用程序仍是使用 C++ 编写的，这不仅是因为更新的语言仍不能满足众多应用程序的需求，还因为 C++ 向程序员提供了灵活性和强大功能。C++ 是一种不断发展的语言，它遵循 ANSI 标准。



### 1.1.1 解释器和编译器

解释器翻译并执行程序，它读取程序指令（源代码），并直接将其转换为操作。编译器先将源代码转换为中间格式，这通常被称为编译，它生成目标文件。然后，编译器调用链接器，将目标文件组合成为可执行程序，其中包含可直接在处理器上运行的机器代码。

由于解释器按原样读取代码并即时执行代码，因此程序员使用起来比较方便。当前，大部分解释型程序被称为脚本，解释器被称为脚本引擎。

编译器增加了一个步骤：将人类能够理解的源代码编译成机器能够理解的目标代码。这个步骤看似不太方便，但由于将源代码转换为机器语言这样一个耗时的任务已经在编译阶段完成，因此编译型程序的运行速度非常快。由于转换工作已经完成，因此执行程序时无需做这样的工作。

诸如 C++ 等编译型语言的另一个优点是，可以向没有编译器的用户提供可执行程序。使用解释型语言时，计算机中必须安装了解释器才能运行程序。

有些语言（如 Visual Basic 6）将解释器称为运行库；其他语言（如 Visual Basic .NET 和 Java）有另一个组件——虚拟机（Virtual Machine, VM）或运行库。VM 也是解释器，但不是源代码解释器——将人类可读的语言转换为依赖于计算机的机器码；而对编译后的独立于计算机的虚拟机语言（中间语言）进行解释和执行。因此，这些语言需要编译器，它负责将程序员编写的源代码进行转换，即将源代码编译成虚拟机或运行库能够解释的内容。

C++ 通常是一种编译型语言，虽然存在一些 C++ 解释器。和众多其他编译型语言一样，C++ 以能够生成快速而功能强大的程序著称。

#### 注意

程序一词通常有两种含义：一种是指程序员编写的指令（源代码），另一种是指可执行软件。这种区分可能会引起巨大的混乱，因此本书尽可能将源代码和可执行文件区分开来。

### 1.1.2 不断变化的需求和平台

当前，程序员们面临的问题与 20 年前完全不同。在 20 世纪 80 年代，人们编写程序是为了管理和处理大量的原始数据。那时，编写代码的人和使用程序的人都是计算机专业人员。现在，越来越多的人开始使用计算机，其中大部分人对计算机和程序的工作原理知之甚少。人们通常只愿意将计算机作为一种工具来解决商务问题，而无心去钻研它。

具有讽刺意味的是，为满足这些新用户易于使用的需求，程序变得越来越复杂。那种需要输入神秘的命令，结果却只看到一大串枯燥的数据的时代已一去不复返。当前的程序大都采用对用户友好的界面，其中包括多个窗口、菜单、对话框以及非常形象的标识，对此我们已经非常熟悉了。

随着万维网的发展，计算机跨入了一个新的市场渗透时代。使用计算机的人比任何时候都多，他们对计算机的期望也非常高。万维网的易用性也提高了人们的期望，他们期望程序能够充分利用万维网提供的信息。

在过去的几年中，应用程序的运行平台已扩展到其他设备，不再限于桌面 PC，而被用于手机、个人数字助理（PDA）、平板 PC 和其他设备。

在本书第一版面世后的几年中，为满足用户的需求，程序员编写的程序变得越来越庞大，越来越复杂。因此，对有助于应对这种复杂性的编程技术的需求显得越来越迫切。

随着编程需求的改变，用于编写程序的语言和技术都得到了发展，以帮助程序员应对这种复杂性。虽然完整的发展历程令人神往，但本书只介绍其中的重要组成部分：从过程化编程到面向对象编程（OOP）的转变。



### 1.1.3 过程化编程、结构化编程和面向对象编程

直到不久前, 计算机程序还被看作是一系列处理数据的过程。过程(函数或方法)是指一组依次执行的指令。数据与过程是分离的, 编程技巧是跟踪哪些函数调用了其他函数以及哪些数据被修改了。为避免这种混乱, 结构化编程应运而生。

结构化编程的主要思想是分而治之。可将计算机程序看作由一系列任务组成。任何过于复杂、无法简单描述的任务都将分解为一系列较小的子任务, 直至每个任务都很小, 很容易理解。

例如, 计算公司职员的平均工资是一项较复杂的任务。然而, 可将其划分成下面几个子任务:

1. 确定公司员工数;
2. 确定每个员工的工资;
3. 计算工资总额;
4. 将工资总额除以员工数。

计算工资总额还可分成以下几步:

1. 读取每个职员的记录;
2. 读取工资额;
3. 将工资额添加到工资总额中;
4. 读取下一个职员的记录。

读取每个职员的记录又可分为以下几步:

1. 打开职员文件;
2. 找到正确的记录;
3. 读取数据。

结构化编程仍是一种非常成功的、处理复杂问题的方法。然而, 到 20 世纪 80 年代后期, 它的许多不足逐渐暴露出来了。

首先, 一种自然而然的愿望是, 将数据(如职员记录)及其操作(排序、编辑等)看作一个整体。不幸的是, 结构化程序将数据结构与处理它们的函数分开, 因此在结构化编程中, 没有将数据同处理它的函数关联起来的自然方式。结构化编程通常被称为过程化编程, 因为其侧重于过程而不是对象。

其次, 程序员们经常发现自己需要重用函数, 但处理一种数据类型的函数通常不能用于处理其他类型的函数, 这限制了重用函数带来的好处。

### 1.1.4 面向对象编程(OOP)

面向对象编程旨在满足以上种种需求, 提供了一种管理极度复杂性的技术, 实现了软件组件的可重用性, 并将数据和操纵数据的任务结合起来。

面向对象编程的实质是模拟对象(东西或概念)而不是数据。要模拟的对象可以是屏幕上的小部件, 如按钮、列表框, 也可以是现实世界中的东西, 如客户、自行车、飞机、猫和水。

对象有特征(属性), 如快、宽敞、黑色、湿; 也有功能, 如购买、加速、飞、叫、冒泡。在编程语言中描述这些对象是面向对象编程的工作。

### 1.1.5 C++和面向对象编程

C++全面支持面向对象编程, 它包括以下三个面向对象开发要素: 封装、继承和多态。

## 1. 封装

工程师需要在正在制造的设备中添加电阻时，通常不会从头开始去制作电阻。他会到电阻库中，检查表示属性的彩色条纹，选择一个满足需求的电阻。对于这位工程师来说，电阻是一个黑盒子——只要电阻满足规格即可，他不关心电阻的内部结构及其工作原理。

能够成为自包容单元的特性称为封装。利用封装可以实现数据隐藏。数据隐藏是一种非常有价值的特性，用户不必了解或关心对象的内部工作原理就可以使用它。就像使用冰箱时不必关心压缩机的工作原理一样，也可以使用设计优良的对象而不必了解其内部工作部件。内部工作部件的变更不会影响程序的运行，条件是它们满足指定的规格，就像可以更换冰箱中的压缩机一样。

同样，工程师在使用电阻时也不必知道电阻的内部状态。电阻的所有属性都封装在电阻对象中，它们不会通过电路传出。无需了解电阻的工作原理就可以有效地使用它。其内部工作部件隐藏在电阻壳内部。

C++通过创建被称为类的用户定义类型来支持封装。第 10 章将介绍如何创建类。类一经创建，就是一个完全封装的实体——一个完整的单元。类的内部工作原理都可以隐藏。对于定义良好的类，用户只需知道如何使用它即可，而不必知道它是如何工作的。

## 2. 继承和重用

Acme 汽车公司的工程师们要制造一款新车时，他们有两种选择：从头做起或修改现有车型。也许他们的 Star 车型已接近完美，但他们想加一个涡轮增压器和一个六速变速装置。总工程师不想从头开始，于是说：“我们另造一款 Star 车吧，但给它增加上述附加装置，并将新型车叫做 Quasar。” Quasar 其实是 Star 车型的一种，只是增加了一些新特征的专用车而已。

C++支持继承。使用继承，可以声明一个新类型作为已有类型的扩展。新子类是从已有类型派生而来的，有时被称为派生类型。例如，如果 Quasar 由 Star 派生而来，它将继承 Star 的所有品质，然后可以增加发动机或对其进行改进。继承及其在 C++中的应用在第 11 章介绍。

## 3. 多态

当您踩下油门时，新的 Quasar 车型可能做出与 Star 车型不同的反应。Quasar 可能喷射燃料并启动涡轮增压器，而 Star 只是让汽油进入化油器中。但对用户而言，他不必要知道这些差别。他只需踩下油门，根据他驾驶的是哪种车，将随之发生正确的响应。

C++支持这种想法，即通过函数多态和类多态，不同的对象将做正确的事情。多态是指同一名称有多种形式，将在第 12 章讨论。

# 1.2 C++的发展历程

当面向对象的分析、设计和编程开始兴起时，Bjarne Stroustrup 对最流行的商用软件开发语言 C 语言进行了扩展，使其可提供面向对象编程所需的功能。

尽管 C++是 C 的超集，且任何合法的 C 程序都是合法的 C++程序，但从 C 跳到 C++的意义是非常重大的。由于 C 程序员都能轻松地进入 C++世界，因此，多年来 C++从其与 C 的关系中获益匪浅。然而，想真正获得 C++带来的好处，很多程序员发现它们不得不放弃很多已有的观念，并学习一种新的分析和解决编程问题的方法。

## 1.3 应该先学习 C 语言吗

既然 C++是 C 的超集，那么学习 C++前是否该先学 C 呢？Stroustrup 和大多数 C++程序员都认为，不仅没有必要先学 C，而且不学可能比学了更好。

C 编程基于结构化编程概念，而 C++ 基于对象化编程。如果先学 C 语言，将不得不忘掉 C 语言提倡的坏习惯。

本书假定读者没有任何编程经验。不过，如果读者是一个 C 程序员，本书的前几章在很大程度上是一种复习。从第 10 章开始将开始介绍真正的面向对象软件开发。

## 1.4 微软的 C++ 托管扩展

在 .NET 中，微软公司新增了 C++ 托管扩展 (Managed Extension)，被称为 Managed C++。这种对 C++ 语言的扩展让 C++ 能够使用微软公司的新平台和函数库。更重要的是，Managed C++ 让 C++ 程序员能够充分利用 .NET 环境的高级功能。要进行专门针对 .NET 平台的开发，除了解有关标准 C++ 的知识外，还必须学习有关这些扩展的知识。

## 1.5 ANSI 标准

隶属于美国国家标准化协会的授权标准委员会 (Accredited Standards Committee) 制定了一个 C++ 国际标准。该 C++ 标准也被称为 ISO (国际标准化组织) 标准、NCITS (国家信息技术标准化委员会) 标准、X3 (NCITS 的前身) 标准和 ANSI/ISO 标准。由于 ANSI 标准使用最广，故本书沿用名称 ANSI 标准。

ANSI 标准旨在确保 C++ 是可移植的，例如，确保为微软的编译器编写的符合 ANSI 标准的代码，用其他厂商的编译器进行编译时不会发生错误。由于本书的代码都符合 ANSI 标准，因此它们都可在 Macintosh、Windows 或 Alpha 计算机上通过编译。

对于大多数 C++ 学习者来说，一般接触不到 ANSI 标准。最新的 C++ 标准为 ISO/IEC 14882-2003，前一版本已稳定使用了很长时间，所有的主要制造商都支持它。我们对本书的所有代码都同 C++ 标准进行了比较，以确保它们符合该标准。

别忘了，并非所有的编译器都全面遵循 C++ 标准。另外，在有些方面，C++ 标准没有做明确的规定，而是留给编译器厂商去选择，因此读者不能完全信任编译器，指望使用不同品牌的编译器进行编译时，程序的运行方式完全相同。

### 注意

由于 C++ 托管扩展只适用于 .NET 平台，并非 ANSI 标准的组成部分，因此本书不介绍这些扩展。

## 1.6 编程准备

所有语言都要求编写程序前对程序作总体设计，C++ 对此要求更高。本书讨论的大多数问题和场景都是通用的，不需要太多设计。然而，对于复杂问题，诸如那些专业程序员每天都会遇到的问题，必须进行设计。设计得越细致，在指定时间和预算内，设计出能够解决指定问题的程序的可能性越大。良好的设计常使程序的错误少且易于维护。据估计，软件成本的 90% 为调试和维护费用。良好的设计能够在很大程度上减少这些费用，对项目的总成本有重要影响。

在准备设计任何程序前，需要问的第一个问题是：要解决什么问题？每个程序都应该有明确的目标，即便是本书中最简单的程序也如此。

非常优秀的程序员要问的第二个问题是：不自己编写软件能完成这些任务吗？重用旧程序、使用纸和笔或者购买现成的软件常常是比编写新程序更好的解决问题的方法。能够提供这种替代办法的程序员永远不愁找不到工作，为今天的问题找到廉价的解决方法总会给以后带来新的机会。

理解了要解决的问题，并确定需要为此编写一个新程序后，便可以开始设计了。彻底搞清问题（分析）并制定解决方案（设计）对编写世界级商业应用程序来说是必不可少的。

## 1.7 开发环境

本书假设您的编译器有一种可以直接写入到控制台的模式（如 MS-DOS 命令提示符或外壳窗口），不用考虑诸如 Windows 和 Macintosh 这样的图形环境。请查找 console 或 easy window 这样的选项或查看编译器文档。

您的编译器可能是集成开发环境（IDE）的一部分，也可能有内置的源代码文本编辑器，或者您可能使用能生成文本文件的商用文本编辑器或字处理器。重要的是，无论在什么环境中编写程序，必须将其保存为简单的纯文本文件，文本中没有嵌入字处理命令。可安全使用的编辑器包括：Windows 记事本、DOS Edit 命令、Brief、Epsilon、Emacs 和 Vi。许多商用字处理软件如 WordPerfect、Word 等也都提供了保存简单文本文件的方法。

使用编辑器创建的文件称为源文件，对于 C++，它们的扩展名通常为 .cpp、.cp 或 .c。本书中所有源代码文件的扩展名都是 .cpp，但请查看您的编译器，看它需要什么样的扩展名。

### 注意

大多数 C++ 编译器并不在乎源代码文件的扩展名，但如果没有指明，许多编译器默认使用 .cpp。然而，需要注意的是，有些编译器将 .c 文件视为 C 代码，而将 .cpp 文件视为 C++ 代码。因此再次提醒您，请查看编译器文档。如果对 C++ 源代码文件统一地使用扩展名 .cpp，其他程序员将能够更容易地了解您的程序。

### 该做

请使用简单的文本编辑器创建源代码或编译器提供的内置编辑器。

保存文件时使用扩展名 .c、.cp 或 .cpp，推荐使用扩展名 .cpp。

查看文档中有关编译器和链接器的信息，以确保您知道如何编译和链接程序。

### 不该做

不要使用保存特殊格式字符的字处理器。如果使用字处理器，必须将文件保存为 ASCII 文本文件。

如果编译器将扩展名为 .c 的文件视为 C 代码而不是 C++ 代码，则不要使用这种扩展名。

## 1.8 创建程序的步骤

创建新程序的第一步是在源文件中编写合适的命令（语句）。尽管源文件中的源代码有点隐晦难懂，任何不懂 C++ 的人都很难理解，但它仍被称为人类可读的格式。源代码文件并不是程序，不能像程序那样执行或运行，但可执行程序文件可以。

### 1.8.1 用编译器生成对象文件

使用编译器可将源代码文件转换为程序。如何启动编译器以及如何告诉它在何处查找源代码随编译器而异，请查看编译器文档。

源代码编译后，将生成一个目标文件，其扩展名通常为 .obj 或 .o，然而它仍不是可执行程序。要把目标文件转换为可执行程序，必须运行链接器。

### 1.8.2 用链接器生成可执行文件

C++ 程序通常是将一个或多个 .obj 文件与一个或多个库链接而生成的。库是编译器提供的一组可链接文件，也可能是单独购买或是程序员创建并编译而成的。所有 C++ 编译器都有一个包含有用函数和



类的库，可以将其包含在程序中。第 6 章和第 10 章将详细介绍函数和类。

创建可执行文件的步骤如下：

1. 创建扩展名为.cpp 的源代码；
2. 将源代码文件编译成扩展名为.obj 或.o 的目标文件；
3. 将目标文件与所需的各种库链接起来，生成可执行程序。

**注意**

如果使用 IDE 进行编程，可能有一个名为“生成”的选项，它将为您完成第 2 步和第 3 步，并提供可执行文件。

## 1.9 程序开发周期

如果每个程序都能在第一运行可行，则完整的程序开发周期为：编写程序、编译源代码、链接程序和运行。遗憾的是，几乎所有程序（无论它多简单）都会有一些错误。有些错误导致无法通过编译，另一些导致无法链接，还有一些仅在运行程序时才会表现出来（这些错误通常被称为 bug）。

无论发现什么类型的错误，都必须改正它，这涉及编辑源代码、重新编译、重新链接以及重新运行程序。图 1.1 描述了程序开发周期中的各个步骤。

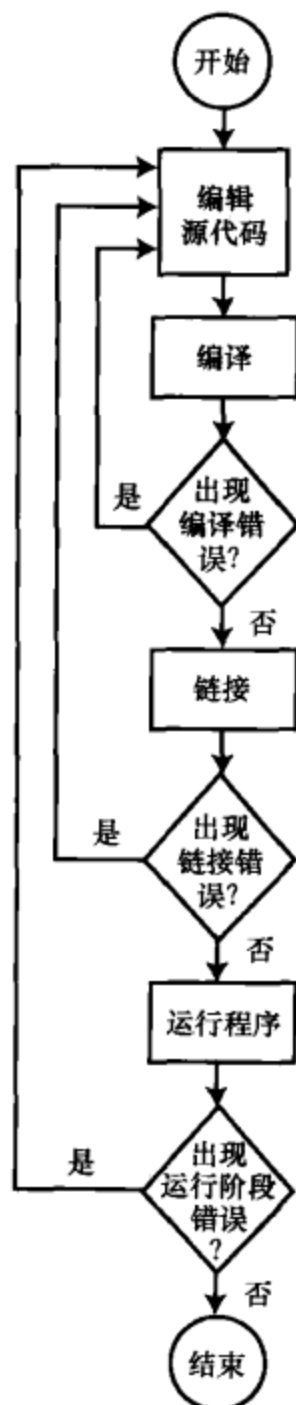


图 1.1 开发 C++ 程序的步骤

## 1.10 HELLO.cpp: 第一个 C++ 程序

传统的编程图书都以在屏幕上打印 Hello World 或其变体开始，本书秉承这个历史悠久的传统。

将程序清单 1.11 中的源代码原封不动（行号除外）地输入到编辑器中，确定输入无误后保存该文件，然后编译、链接并运行它。如果一切顺利，它将在屏幕上打印 Hello World。现在不必过多考虑该程序的工作原理，这里提供它旨在让读者熟悉开发周期。接下来的几章将详细介绍该程序的各个方面。

### 警告

在下面的程序清单中，左边的行号旨在方便书中引用，不应将它们输入到编辑器中。例如，对于清单 1.1 的第 1 行，应该输入：

```
#include <iostream>
```

### 程序清单 1.1 HELLO.cpp

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!\n";
6:     return 0;
7: }
```

请务必照原样输入，特别要注意标点符号。第 5 行中的 << 是重定向符，在大多数键盘上可按下 Shift 键，再按两次逗号键输入。第 5 行中的 std 和 cout 之间是 2 个冒号 (::)。第 5 行和第 6 行都以分号 (;) 结束。

另外，务必正确地使用编译器。大多数编译器都会自动链接，但请查看文档，看是否需要提供特殊的选项或执行命令来进行链接。

如果遇到错误，请仔细检查代码，看它与上面的程序清单有何不同。如果第 1 行有错，如 cannot find file iostream，请查看编译器文档，了解如何设置包含路径或环境变量。

如果出现指出没有 main 函数原型的错误，请在第 3 行前加一行：int main( );（这是有些编译器使用的 main 函数原型）。在这种情况下，必须在本书中每个程序的 main 函数前添加上述代码行。大多数编译器都无此要求，只有少数编译器要求这样做。如果您的编译器是这样的，完成后的程序应如下：

```
#include <iostream>
int main();           // most compilers don't need this line
int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

### 注意

如果不知道特殊字符和关键字的发音，将难以阅读程序。第 1 行应读做“英镑符号 include eye-oh-stream”。第 5 行应读做“ess-tee-dee-see-out Hello World”。

在 Windows 系统上，试着运行 HELLO.exe（或在您的操作系统中运行相应的可执行文件。例如，在 UNIX 系统上，应运行 HELLO，因为在 UNIX 中可执行程序没有扩展名）。它将 Hello World! 直接写到屏幕上。如果情况果真如此，那么祝贺您！您输入、编译并运行了您的第一个 C++ 程序。这好像没什么了不起，但几乎每位专业 C++ 程序员都是从这个简单程序起步的。

使用 IDE（如 Visual Studio 或 Borland C++ Builder）的程序员可能发现，运行该程序时，将弹出一个窗口，然后迅速消失，根本没机会看清程序的输出。在这种情况下，请在源代码的 return 语句前添



加下述代码行：

```
char response;  
std::cin >> response;
```

这些代码行导致程序暂停，直到用户输入一个字符（可能还需要按下回车键）。它们让用户有机会看到运行结果。如果对于程序 `hello.cpp` 需要这样做，则对于本书中的大部分程序也需要这样做。

#### 使用标准库

如果使用的是非常旧的编译器，上述程序将无法运行——找不到新的 ANSI 标准库。在这种情况下，请按如下修改该程序：

```
1: #include <iostream.h>  
2:  
3: int main()  
4: {  
5:     cout << "Hello World!\n";  
6:     return 0;  
7: }
```

注意，现在库名以 `.h` 结尾，在第 5 行不再在 `cout` 前使用 `std::`。这是 ANSI 标准之前的老式头文件样式。如果您的编译器采用这种方式，而不是前面的方式，则是过时的。为编译本书的示例，读者应更新编译器或下载带最新 C++ 编译器的免费 IDE。

## 1.11 编译器初步

82903-270-9336912-75850

本书不针对特定的编译器。这意味着本书的程序可以在任何平台（Windows、Mac、UNIX、Linux 等）上使用任何遵循 ANSI 标准的 C++ 编译器进行编译。

当前，大多数程序员在 Windows 环境下编程，大多数专业程序员使用微软编译器。这里无法介绍每种编译器的编译和链接细节，但将介绍如何使用 Visual C++ 6，这种编译器的用法与其他编译器相似。

然而，由于编译器的差异，请务必查看编译器文档。

### 创建 Hello World 项目

要创建并测试 Hello World 程序，请按以下步骤执行：

1. 打开编译器；
2. 从菜单中选择 File/New；
3. 选择 Win32 Console Application，然后输入项目名，如 `hello`，最后单击 OK 按钮；
4. 从选项菜单中选择 An Empty Project，单击 Finish 按钮；将出现一个对话框，其中包含新项目的信息；
5. 单击 OK 按钮，回到主编辑器窗口；
6. 从菜单中选择 File/New；
7. 选择 C++ Source File，并在文本框 File Name 中输入一个名称，如 `hello`；
8. 单击 OK 按钮，回到主编辑器窗口；
9. 输入前面所示的代码；
10. 选择菜单 Build/Build `hello.exe`；
11. 确保没有编译错误，这种信息出现在编辑器底部附近；
12. 按 `Ctrl+F5` 或选择菜单 Build/Execute `hello` 来运行该程序；
13. 按空格键结束程序。

**FAQ**

我能够运行该程序，但窗口稍瞬即逝，无法阅读运行结果。请问哪里出现了问题？

答：查看编译器文档，应该有让程序暂停的方法。对于微软编译器，方法是按 Ctrl + F5。

对于其他编译器，可以在返回语句之前（即程序清单 1.1 的第 5 行和第 6 行之间）添加如下代码：

```
char response;  
std::cin >> response;
```

这将导致程序暂停，等待用户输入一个字符。要结束程序，可输入任何字母或数字（如 1），然后按回车键（如果必要的话）。

std::cout 和 std::cin 的含义将在第 2 章和第 27 章讨论，现在只需将其作为神奇的咒语使用即可。

## 1.12 编译错误

发生编译错误的原因有很多，通常是由于输入错误或其他不经意的小错误引起的。优秀的编译器不仅能告诉您发生了什么错误，还会准确指出代码中的出错位置。更好一些的编译器甚至能指出修改方法。

可以故意在程序中加入一个错误来验证这一点。如果 HELLO.cpp 运行正常，请编辑它并把程序清单 1.1 中第 7 行的括号去掉，修改后的程序如程序清单 1.2 所示。

程序清单 1.2 演示编译错误

```
1: #include <iostream>  
2:  
3: int main()  
4: {  
5:     std::cout << "Hello World!\n";  
6:     return 0;
```

重新编译该程序，您将看到类似下面这样的错误消息：

```
Hello.cpp(7) : fatal error C1004: unexpected end of file found
```

错误消息指出有问题的文件和行号以及是什么问题（尽管有点难懂）。在这里，错误消息指出，找遍整个源代码文件，直到到达末尾也没有找到右大括号。

有时候，错误消息只指出错误的大概位置。如果编译器能够准确地识别每个问题，将能够修复代码。

## 1.13 总结

阅读本章后，读者应该对 C++ 的发展历程及其用于解决什么问题有深入的认识。读者应该深信，对任何对编程感兴趣的人来说，学习 C++ 都是正确的选择。C++ 提供了面向对象编程工具及系统级语言性能，这使其成为当今开发语言的最佳选择。

在本章中，您学习了如何输入、编译、链接和运行第一个 C++ 程序，并了解了程序开发周期。您还对什么是面向对象编程有一定的了解。在接下来的章节中，您还将不断学习这些主题。

## 1.14 问与答

问：文本编辑器和字处理器之间有何区别？

答：文本编辑器生成的文件是纯文本文件，没有字处理器所需的格式化命令或特殊符号。简单文本编辑器不支持自动换行、粗体、斜体等。

问：如果编译器有内置的编辑器，必须使用它吗？

答：几乎所有的编译器都能编译由任何文本编辑器生成的代码。然而，使用内置文本编辑器的好处是，在开发周期的编辑和编译步骤之间可以快速地来回移动。高级编译器包含一个完全集成的开发环境，让程序员能够访问帮助文件、就地编辑和编译代码，同时不必离开开发环境即可解决编译和链接错误。

问：可以忽略编译器发出的警告消息吗？

答：编译器通常会指出错误和警告。如果有错误，程序将不能通过编译。如果只有警告，编译器通常是创建可执行程序。

很多书籍对这个问题都含糊其辞。正确的答案是不能！一开始就应养成将警告消息视为错误的习惯。当您所做之事并非您本意时，C++将通过编译器向您发出警告。请认真对待这些警告并设法消除它们。有些编译器甚至有一个设置选项，将所有警告视为错误，进而禁止生成可执行程序。

问：什么是编译阶段？

答：编译阶段是指运行编译器的阶段，与链接阶段（运行链接器时）和运行阶段（运行程序时）相对。这只是程序员用来表示错误通常出现的三种阶段的简称。

## 1.15 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 1.15.1 测验

1. 解释器和编译器有何不同？
2. 如何使用编译器编译源代码？
3. 链接器的作用是什么？
4. 开发周期包括哪些步骤？

### 1.15.2 练习

1. 阅读下面的程序，不运行它的情况下猜测其功能：

```
1: #include <iostream>
2: int main()
3: {
4:     int x = 5;
5:     int y = 7;
6:     std::cout << std::endl;
7:     std::cout << x + y << " " << x * y;
8:     std::cout << std::endl;
9:     return 0;
10: }
```

2. 输入练习 1 中的程序，然后编译并链接它。它做什么？与您的猜测相符吗？
3. 输入并编译下面的程序。发生了什么错误？

```
1: include <iostream>
2: int main()
3: {
4:     std::cout << "Hello World \n";
5:     return 0;
6: }
```

4. 修复练习 3 中程序的错误，重新编译、链接并运行它。它做什么？



## 第 2 章

# C++程序的组成部分

C++程序由类、函数、变量及其他元素组成。本书的大部分内容对这些组成部分进行深入解释，但为了解程序是如何组合在一起的，必须分析一个完整的工作程序。

在本章中，您将学习：

- C++程序的组成部分
- 各部分如何协同工作
- 函数及其用途

### 2.1 一个简单程序

即使是第 1 章的简单程序 HELLO.cpp 也有很多有趣的组成部分。本节将对该程序进行更详细的分析。为方便起见，程序清单 2.1 重新列出了 HELLO.cpp 的最初的版本。

程序清单 2.1 用 HELLO.cpp 说明 C++程序的组成部分

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!\n";
6:     return 0;
7: }
```

#### ▼ 输出：

Hello World!

#### ▼ 分析：

在第 1 行，文件 `iostream` 被包含到当前文件中。下面详细分析第 1 行：第 1 个字符是符号 `#`，这是预处理器标记。每次启动编译器时，将首先运行预处理器。预处理器浏览源代码，查找以 `#` 开头的行，在编译器运行之前处理这些行。有关预处理器的详细内容请参阅第 15 章和第 29 章。

`include` 是一条预处理器指令，指出接下来是一个文件名，请找到该文件，读取它并将其放到这里。文件名两边的尖括号告诉预处理器，在通常的所有位置查找该文件。如果编译器安装正确，尖括号将使预处理器在保存编译器的所有包含文件的目录下查找 `iostream` 文件。文件 `iostream`（输入输出流）由 `cout` 使用，用来向屏幕输出。第 1 行的作用就是将 `iostream.h` 文件包含到该程序中，就好像是您自己亲自将其输入一样。

#### 注意

每次调用编译器时预处理器总是在编译器之前运行。预处理器把任何以 `#` 开头的行都翻译成一条特殊命令，为编译器准备好代码文件。

**注意**

并非所有的编译器都支持在#include语句中省略文件扩展名。如果出现错误消息,则可能需要修改编译器的包含搜索路径或在#include语句中加上文件扩展名。

实际程序从函数main()开始。每个C++程序都有一个main()函数。函数是执行一项或多项操作的代码块。通常函数是由其他函数调用的,但main()特殊。程序在开始时会自动调用main()。

和所有函数一样,main()函数也必须指出返回值类型。在HELLO.cpp中,main()的返回值类型为int,这意味着在其运行结束后将向操作系统返回一个整数。在这个程序中,它返回整数0。可向操作系统返回一个指出成功还是失败的值,也可使用故障编码描述失败原因。其他应用程序可根据“退出编码”判断应用程序是否成功执行。

**注意**

有些编译器允许将main()的返回类型声明为void。这已不再是合法的C++,请不要养成这种坏习惯。让main()返回int,只需返回0即可,就像上面main()中最后一行那样。

**注意**

有些操作系统允许用户测试程序的返回值。习惯做法是返回0,表示程序正常结束。

所有函数都以左大括号({)开始,以右大括号(})结束。main()函数的两个大括号分别在第4行和第7行。在两个大括号之间的所有代码均被看作是函数的一部分。

std::cout是该程序的灵魂。对象cout用于将一条消息打印到屏幕上。第10章将简要介绍对象,而cout和cin将在第27章详细介绍。在C++中,这两个对象(cin和cout)分别用来处理输入(如从键盘输入)和输出(如输出到控制台)。

cout是标准库提供的一个对象。库是一组类。标准库遵循ANSI标准的每个C++编译器都提供的一组标准类。

您告诉编译器,您要使用的cout对象位于标准库中,这是使用名称空间说明符std指出的。由于可能有来自多个厂商的同名对象,因此C++将世界划分为名称空间。名称空间是一种分类方法,例如,如果说到cout时,指的是标准名称空间而不是其他名称空间中的cout,可以使用std::cout向编译器指出这一点。接下来的几章将详细地讨论名称空间。

cout的用法如下:输入cout和输出重定向运算符(<<)。输出重定向运算符后面的所有内容都被输出到屏幕上。要输出一个字符串,请务必使用双引号(")将其括起,如程序清单2.1所示。

**注意**

需要注意的是,重定向运算符由两个小于号组成,它们之间没有空格。

文本字符串是一系列可打印的字符。

最后两个字符(\n)告诉cout,在Hello World!之后换行。这个特殊代码将在第18章讨论cout时详细介绍。

main()函数以右大括号(})结束。

## 2.2 cout简介

在第27章,您将学习如何使用cout将数据打印到屏幕。就现在而言,您可直接使用它,而不必关心其工作原理。要将一个值输出到屏幕,只需写cout,后面跟上插入运算符(<<)。插入运算符(<<)由两个小于号(<)组成,但C++将其作为一个字符看待。

最后在插入运算符后输入要输出的数据。程序清单2.2演示了如何使用cout。准确地按程序清单输入,并将其中的Jesse Liberty改成您的名字。当然,如果您也叫Jesse Liberty,那就不用了。

## 程序清单 2.2 使用 cout

```
1: // Listing 2.2 using std::cout
2: #include <iostream>
3: int main()
4: {
5:     std::cout << "Hello there.\n";
6:     std::cout << "Here is 5: " << 5 << "\n";
7:     std::cout << "The manipulator std::endl ";
8:     std::cout << "writes a new line to the screen.";
9:     std::cout << std::endl;
10:    std::cout << "Here is a very big number:\t" << 70000;
11:    std::cout << std::endl;
12:    std::cout << "Here is the sum of 8 and 5:\t";
13:    std::cout << 8+5 << std::endl;
14:    std::cout << "Here's a fraction:\t\t";
15:    std::cout << (float) 5/8 << std::endl;
16:    std::cout << "And a very very big number:\t";
17:    std::cout << (double) 7000 * 7000 << std::endl;
18:    std::cout << "Don't forget to replace Jesse Liberty ";
19:    std::cout << "with your name...\n";
20:    std::cout << "Jesse Liberty is a C++ programmer!\n";
21:    return 0;
22: }
```

## ▼ 输出:

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here is the sum of 8 and 5:     13
Here's a fraction:              0.625
And a very very big number:     4.9e+007
Don't forget to replace Jesse Liberty with your name...
Jesse Liberty is a C++ programmer!
```

## 警告

有些编译器存在错误，要求在加法操作数两边加上圆括号，即将第 13 行改为：  
`cout << (8+5) << std::endl;`

## ▼ 分析:

语句 `#include<iostream>` 将 `iostream` 文件加入到源代码中。使用 `cout` 及其相关函数时必须这样做。

该程序首先以最简单的方式使用 `cout`：打印字符串（一系列字符）。`\n` 是一个特殊的格式符，它告诉 `cout` 另起一行。

在第 6 行中，将 3 个值传递给了 `cout`：

```
std::cout << "Here is 5: " << 5 << "\n";
```

这些值由插入运算符 (`<<`) 分开。第 1 个值是字符串 `Here is 5:`。请注意冒号后的空格，它也是字符串的组成部分。接下来，将 5 传递给插入运算符，然后传递换行符（总是在双引号或单引号中）。这将导致将下述内容打印到控制台：

```
Here is 5: 5
```

由于在第 1 个字符串后没有换行符，因此第 2 个值紧接着第 1 个值打印。这被称为串接两个值。

注意到这里使用了控制符 `std::endl`。`endl` 的作用是另起一行，因此与 `'\n'` 等效。`endl` 也由标准库提供，因此在它前面加上了 `std::`，就像在 `cout` 前面要加上 `std::` 一样。

## 注意

`endl` 表示 end line，是 end-ell 而不是 end-one。通常它念作 “end-ell”。

`endl` 比 `"\n"` 更好，因为 `endl` 将适应当前使用的操作系统，而在有些 OS 或平台上，“`\n`”可能不是完整的换行符。

格式化字符 `\t` 插入一个制表符，该程序的其他行演示了如何使用 `cout` 显示整数、小数等。第 10 行



至第 16 行使用它来将输出对齐。第 10 行表明, 不仅可打印整型数据, 也可打印长整型数据。第 13 行和第 14 行表明, `cout` 还可进行简单加法。在第 14 行中, 将 `8+5` 的值传递给 `cout`, 而输出是 13。(float) 和(double)命令 `cout` 将数字显示为浮点数。第 3 章讨论数据类型时, 将介绍这些数据类型。

应用自己的名字代替 Jesse Liberty, 这样, 输出结果将确认您是一个真正的 C++ 程序员。这肯定是真的, 因为计算机是这么说的。

## 2.3 使用标准名称空间

您将发现, 在 `cout` 和 `endl` 前面使用 `std::` 非常烦人。尽管使用名称空间指示是一种很好的方式, 但大量的输入很讨厌。ANSI 标准提供了两种方法来解决这个小问题。

第一种方法是, 在代码清单的开头告诉编译器, 您将使用标准库 `cout` 和 `endl`, 如程序清单 2.3 的第 5 行和第 6 行所示。

程序清单 2.3 使用关键字 `using`

---

```

1: // Listing 2.3 - using the using keyword
2: #include <iostream>
3: int main()
4: {
5:     using std::cout; // Note this declaration
6:     using std::endl;
7:
8:     cout << "Hello there.\n";
9:     cout << "Here is 5: " << 5 << "\n";
10:    cout << "The manipulator endl ";
11:    cout << "writes a new line to the screen.";
12:    cout << endl;
13:    cout << "Here is a very big number:\t" << 70000;
14:    cout << endl;
15:    cout << "Here is the sum of 8 and 5:\t";
16:    cout << 8+5 << endl;
17:    cout << "Here's a fraction:\t\t";
18:    cout << (float) 5/8 << endl;
19:    cout << "And a very very big number:\t";
20:    cout << (double) 7000 * 7000 << endl;
21:    cout << "Don't forget to replace Jesse Liberty ";
22:    cout << "with your name...\n";
23:    cout << "Jesse Liberty is a C++ programmer!\n";
24:    return 0;
25: }
```

---

### ▼ 输出:

---

```

Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here is the sum of 8 and 5:      13
Here's a fraction:              0.625
And a very very big number:      4.9e+007
Don't forget to replace Jesse Liberty with your name...
Jesse Liberty is a C++ programmer!
```

---

### ▼ 分析:

输出与前一个程序清单相同。程序清单 2.3 和 2.2 的唯一差别是: 在第 5 行和第 6 行, 添加了告诉编译器将使用标准库中两个对象的语句, 这是使用关键字 `using` 来完成的。这样, 就不再需要限定 `cout` 和 `endl` 对象了。

第二种解决办法是, 告诉编译器程序将使用整个标准名称空间; 也就是说, 除非特别说明, 否则任何对象都来自标准名称空间。在这种情况下, 应使用 `using namespace std;` 而不是 `using std::cout;`, 如程序清单 2.4 所示。

程序清单 2.4 使用关键字 namespace

```
1: // Listing 2.4 - using namespace std
2: #include <iostream>
3: int main()
4: {
5:     using namespace std; // Note this declaration
6:
7:     cout << "Hello there.\n";
8:     cout << "Here is 5: " << 5 << "\n";
9:     cout << "The manipulator endl ";
10:    cout << "writes a new line to the screen.";
11:    cout << endl;
12:    cout << "Here is a very big number:\t" << 70000;
13:    cout << endl;
14:    cout << "Here is the sum of 8 and 5:\t";
15:    cout << 8+5 << endl;
16:    cout << "Here's a fraction:\t\t";
17:    cout << (float) 5/8 << endl;
18:    cout << "And a very very big number:\t";
19:    cout << (double) 7000 * 7000 << endl;
20:    cout << "Don't forget to replace Jesse Liberty ";
21:    cout << "with your name...\n";
22:    cout << "Jesse Liberty is a C++ programmer!\n";
23:    return 0;
24: }
```

### ▼ 分析:

同样, 输出也与该程序的以前版本相同。使用 `using namespace std;` 的优点是, 不再需要指定实际使用的对象 (如 `cout` 和 `endl`); 缺点是可能不小心地使用了错误库中的对象。

纯正论者喜欢在 `cout` 和 `endl` 的前面使用 `std::`; 而懒人喜欢使用 `using namespace std;`。在本书中, 大部分情况下将使用 `using` 来声明要使用的名称, 但出于好玩, 偶尔也会使用一下另一种风格。

## 2.4 对程序进行注释

当您编写程序时, 您的意图对您来说是清晰和不言而喻的。有趣的是, 一个月后, 您再来看您的程序, 将发现令人迷惑。没有人知道为何会这样, 但这种事情经常发生。

为避免自己迷惑, 同时帮助他人理解您的程序, 应使用注释。注释是被编译器忽略的文本, 但可以告诉读者, 程序的某个地方想做什么。

### 2.4.1 注释的类型

C++ 注释有两种: 单行注释和多行注释。

单行注释使用双斜杠 (`//`) 来表示。双斜杠告诉编译器, 忽略之后到行尾的所有内容。

多行注释以斜杠和星 (`/*`) 打头。这种注释标记告诉编译器, 忽略之后到星号和斜杠 (`*/`) 之间的所有内容。这两种注释标记可以位于同一行, 它们之间也可以有一行或多行, 但每个 `/*` 必须有与之匹配的 `*/`。

很多 C++ 程序员在大多数时候使用双斜杠型单行注释, 多行注释标记用于将程序中的大型代码块注释掉。在使用多行注释标记注释掉的代码块中, 可以包含单行注释。多行注释标记之间的所有内容 (包括双斜杠型注释) 都将被忽略。

#### 注意

多行注释被称为 C 风格注释, 因为它是 C 编程语言引入和使用的。单行注释最初是 C++ 而不是 C 语言的组成部分, 因此被称为 C++ 风格注释。最新的 C 和 C++ 标准都支持这两种注释风格。

## 2.4.2 使用注释

有些人建议在函数的开头编写注释，说明函数的功能和返回值。函数应该有一个明确说明其功能的名称，那些令人迷惑的代码都应重新设计和重新编写，使其含义是不言而喻的。不应将注释用作编写晦涩代码的借口。

这并不是说绝不要使用注释，只是不应依赖注释来阐明晦涩的代码，而应修改这样的代码。总之，应编写良好的代码，将注释作为解释代码的辅助工具。

程序清单 2.5 演示注释的用法，表明了注释对程序的运行和结果没有任何影响。

程序清单 2.5 演示注释用法的 HELP.cpp

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using std::cout;
6:
7:     /* this is a comment
8:      and it extends until the closing
9:      star-slash comment mark */
10:    cout << "Hello World!\n";
11:    // this comment ends at the end of the line
12:    cout << "That comment ended!\n";
13:
14:    // double-slash comments can be alone on a line
15:    /* as can slash-star comments */
16:    return 0;
17: }
```

### ▼ 输出:

```
Hello World!
That comment ended!
```

### ▼ 分析:

第 7~9 行以及第 11、14、15 行的注释都被编译器忽略。第 11 行的注释到行尾结束，而第 7 行和第 15 行的注释需要注释结束标记。

### 注意

有些编译器支持第三种注释方式，这种注释被称为文档注释，用三个斜杠 (///) 标记。支持这种注释的编译器让您能够使用这些注释生成有关程序的文档。由于这种注释当前不是 C++ 标准的组成部分，因此这里不介绍它。

## 2.4.3 有关注释的警告

对明显的事情进行注释往往没有多大意义。事实上，注释可能降低效率，因为代码可能被修改，而程序员往往忘记修改注释。然而，一个人觉得很明显的事情，在另一个人看来也许是晦涩的，因此在什么情况下添加注释需要一定的判断力。

通用规则是，注释不应说明发生了什么事情，而应说明为什么会发生这种事情。

## 2.5 函数

尽管 `main()` 是一个函数，但它并不是通常意义上的函数。函数只有在程序执行时被调用才能起作用，而 `main()` 函数由操作系统调用。

通常，如果没有遇到函数，程序将依次逐行地执行源代码。遇到函数后，程序将执行该函数。函数结束后，程序又返回到调用该函数代码行的下一行继续执行。

函数的作用可用削铅笔来比喻。画图时，如果铅笔突然断了，您可能先停止画图去削铅笔，削好铅笔后再继续画图。程序需要某项服务时，可以调用一个函数来执行这项服务，待函数执行完后，再返回到原来的地方继续执行。程序清单 2.6 说明了这一点。

#### 注意

第 6 章将更详细地介绍函数；函数可返回的数据类型将在第 3 章详细介绍。本章只简要地介绍函数，因为几乎所有的 C++ 程序都需要用到函数。

程序清单 2.6 演示函数调用

```
1: #include <iostream>
2:
3: // function Demonstration Function
4: // prints out a useful message
5: void DemonstrationFunction()
6: {
7:     std::cout << "In Demonstration Function\n";
8: }
9:
10: // function main - prints out a message, then
11: // calls DemonstrationFunction, then prints out
12: // a second message.
13: int main()
14: {
15:     std::cout << "In main\n" ;
16:     DemonstrationFunction();
17:     std::cout << "Back in main\n";
18:     return 0;
19: }
```

#### ▼ 输出:

```
In main
In Demonstration Function
Back in main
```

#### ▼ 分析:

第 6~8 行定义了函数 `DemonstrationFunction()`。被调用时，它将一条消息打印到屏幕然后返回。

实际程序开始于第 13 行。在第 15 行中，`main()` 函数打印了一条消息，指出当前位于 `main()` 函数中。打印这条消息后，第 16 行调用 `DemonstrationFunction()` 函数。该调用导致程序流程跳转到第 5 行的函数 `DemonstrationFunction()`，进而执行该函数中的所有命令。在这个程序中，整个函数由第 7 行的代码组成，它打印另一条消息。`DemonstrationFunction()` 执行完毕后（第 8 行），程序返回到函数调用的下一行，这里为第 17 行。在这一行，`main()` 打印最后一条消息。

### 2.5.1 使用函数

函数要么返回一个值，要么返回 `void`，后者意味着什么也不返回。将两个整数相加的函数可能返回它们的和，因此需要将其定义为返回一个整型值。如果函数只打印一条消息而不返回任何东西，应将其定义为返回 `void`。

函数由函数头和函数体组成，而函数头又由返回类型、函数名和参数组成。函数参数让您能够将值传递给函数。因此，如果函数将两个数相加，则这两个数都作为函数的参数。下面是一个典型的函数头，它声明了一个名为 `Sum` 的函数，该函数接受两个整数值（`first` 和 `second`），并返回一个整数值：

```
int Sum( int first, int second)
```

参数用于声明要传入的值的类型；调用函数时实际传入的值被称为实参。很多程序员都将参数和实参看做同义词；另一些人则将它们分开。对于 C++ 编程而言，参数和实参之间的区别并不重要，因此看到这两个词被交替使用时，读者不用担心。

函数主体由左大括号、零条或更多的语句以及右大括号组成。函数的功能由语句实现。

函数可能使用 `return` 语句来返回一个值。返回的值必须是函数头中声明的类型。另外，该语句导致函数结束。如果函数中不包括 `return` 语句，它将在结束时自动返回 `void`。如果函数被声明为返回一个值，但没有 `return` 语句，有些编译器将发出警告或错误消息。

程序清单 2.7 演示了一个接受两个整型参数并返回一个整型值的函数。读者不必考虑其中语法以及如何使用整型值（如 `int first`）的细节，这些内容将在第 3 章详细介绍。

程序清单 2.7 一个简单函数

---

```

1: #include <iostream>
2: int Add (int first, int second)
3: {
4:     std::cout << "Add() received "<< first << " and "<< second <<
    "\n";
5:     return (first + second);
6: }
7:
8: int main()
9: {
10:    using std::cout;
11:    using std::cin;
12:
13:
14:    cout << "I'm in main()!\n";
15:    int a, b, c;
16:    cout << "Enter two numbers: ";
17:    cin >> a;
18:    cin >> b;
19:    cout << "\nCalling Add()\n";
20:    c=Add(a,b);
21:    cout << "\nBack in main().\n";
22:    cout << "c was set to " << c;
23:    cout << "\nExiting...\n\n";
24:    return 0;
25: }
```

---

#### ▼ 输出:

```

I'm in main()!
Enter two numbers: 3 5

Calling Add()
In Add(), received 3 and 5

Back in main().
c was set to 8
Exiting...
```

#### ▼ 分析:

第 2 行定义了 `Add()` 函数。它接受两个整型参数并返回一个整型值。程序本身从第 8 行开始。在第 16 行，程序提示用户输入两个数。用户输入两个数（用空格将两数分开），然后按回车键。用户输入的数字被存储在第 17 行和第 18 行的变量 `a` 和 `b` 中。在第 20 行，`main()` 将用户输入的两个数作为参数传递给 `Add()` 函数。

程序跳到从第 2 行开始的 `Add()` 函数处执行。该函数通过参数 `first` 和 `second` 收到存储在 `a` 和 `b` 中的值，然后将它们打印并相加。第 5 行返回结果，同时返回到调用它的函数——这里为 `main()`。

在第 17 行和第 18 行，对象 `cin` 被用来获取变量 `a`、`b` 的值；在程序余下的地方，`cout` 被用来输出到控制台。变量及该程序的其他方面将在下一章深入介绍。

## 2.5.2 方法和函数

无论使用何种称呼，函数始终是函数。需要注意的是，不同的语言和编程方法学可能使用不同的术语来称呼函数。一个较常用的术语是方法。方法是类中函数的另一种称呼。

## 2.6 总结

学习复杂主题（如编程）的困难在于，正在学习的东西又依赖于其他有待学的东西。本章介绍了C++程序的基本组成部分。

## 2.7 问与答

问：#include 的作用是什么？

答：这是一个预处理器编译指令。预处理器在您调用编译器时运行。该指令使得预处理器将 include 后面的<>中的文件读入程序，其效果如同将这个文件输入到源代码中的这个位置。

问：//注释和/\*注释之间有何不同？

答：//注释到行尾结束；/\*注释到\*/结束。//注释也被称为单行注释，/\*注释通常被称为多行注释。请记住，即使是函数的结尾也不能作为/\*注释的结尾，必须加上注释结尾标记\*/，否则将出现编译阶段错误。

问：好注释与坏注释的区别在哪里？

答：好注释告诉读者某段代码是如何工作的或它将要做什么；而坏注释只说明某行代码做什么。代码行的含义应不言而喻，良好的代码行应该不用注释就能让读者明白其功能。

## 2.8 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录D的答案，继续学习下一章前，请务必弄懂这些答案。

### 2.8.1 测验

1. 编译器和预处理器有何不同？
2. 为什么函数 main() 是特殊的？
3. 有哪两种注释？它们有何不同？
4. 注释能否嵌套？
5. 注释可以长于一行吗？

### 2.8.2 练习

1. 编写一个程序，将“I love C++”打印到控制台。
2. 编写一个可以编译、链接和运行的最短程序。



3. 查错：输入下面的程序并编译它。它为什么不能通过编译？如何修复？

```
1: #include <iostream>
2: main()
3: {
4:     std::cout << "Is there a bug here?";
5: }
```

4. 修复练习 3 中的错误，然后重新编译、链接并运行它。

5. 修改程序清单 2.7，以包含一个减法函数。将该函数命名为 `Subtract()`，并像使用 `Add()` 函数那样使用它。另外，应将传递给函数 `Add()` 的值传递给该函数。

## 第3章

# 使用变量和常量

程序需要一种方式来存储其使用或创建的数据，以便在后面的程序执行期间能够使用它们。变量和常量提供各种表示、存储和操纵数据的方式。

在本章中，您将学习：

- 如何声明和定义变量与常量
- 如何给变量赋值以及操纵这些值
- 如何将变量的值显示到屏幕上

### 3.1 什么是变量

在 C++ 中，变量是储存信息的地方。变量是计算内存中的一个位置，可以在其中存储值或检索其中的值。

变量用于临时存储，退出程序或关机后，变量中的信息将丢失。永久存储则不同，变量的值被永久地存储到数据库或磁盘文件中。第 27 章将讨论存储到磁盘文件中。

#### 3.1.1 将数据存储在内存中

可将计算机内存看作是一系列文件柜，每个文件柜由许多排成一系列的小格子组成。每个文件柜（内存单元）都按顺序进行编号。这些编号被称为内存地址。变量通常预留一个或多个文件柜，可以在其中存储一个值。

变量名（如 `myVariable`）是贴在文件柜上的标签，这样无需知道变量的地址，就可方便地找到变量。图 3.1 说明了这一点。从中可知，变量 `myVariable` 从内存地址 103 开始。根据 `MyVariable` 的大小，它可能占用一个或多个内存地址。

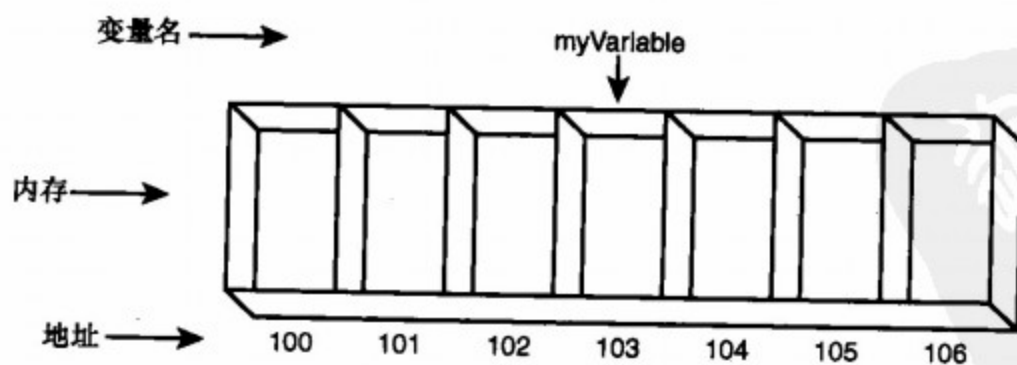


图 3.1 内存示意图

#### 注意

RAM 是指随机存取存储器（Random Access Memory）。运行程序时，从磁盘文件将程序加载到 RAM 中。所有变量都是在 RAM 中创建的。程序员谈到内存时，通常指的是 RAM。

### 3.1.2 预留内存

在 C++ 中定义变量时，必须告诉编译器该变量的类型：整数、浮点数、字符型等。这种信息告诉编译器预留多大空间以及您将在变量中存储什么样的值；它还让编译器能够在您将类型不对的值存储到变量中时发出警告或错误消息，编程语言的这种特征被称为类型检查 (strong typing)。

每个文件格的大小都是 1 字节。如果您创建的变量占 4 字节，则需要 4 字节内存，也就是 4 个文件格。变量类型（如整型）告诉编译器给变量预留多大内存（多少个文件格）。

有时候，程序员必须理解“位”和“字节”的含义，毕竟这些是基本的存储单位。虽然计算机程序在剔除这些细节方面越来越强，但了解数据是如何存储的仍有帮助。有关二进制数学中底层概念的简介，请参阅附录 A。

#### 注意

如果数学让您厌烦，就不要学习附录 A，您并非真正需要这方面的知识。实际上，虽然熟悉逻辑和理性思维很重要，但程序员不再需要是数学家。

### 3.1.3 整型变量的大小

在任何计算机中，每种变量都占据一定的内存。也就是说，整数在一台机器上可能是 2 字节，而在另一台机器上可能是 4 字节，但对任何一台机器而言，这个值是固定的，不随时间而变化。

单个字符（包括字母、数字和符号）用 char 变量存储，这种变量通常为 1 字节。

对于较小的整数，可以使用 short 变量来存储。在大多数计算机上，short 变量为 2 字节，long 变量为 4 字节，而 int 变量（没有关键字 short 或 long）通常为 2 或 4 字节。

读者可能认为，语言规定了每种类型的长度，但 C++ 没有这样做，它只这样规定：short 的长度不能超过 int，而 int 的长度不能超过 long。也就是说，在您使用的计算机上，short 可能为 2 字节，而 int 和 long 都是 4 字节。

int 的长度由处理器（16 位、32 位或 64 位）和编译器决定。在使用现代编译器和 Intel 奔腾处理器的 32 位计算机上，int 为 4 字节。

#### 注意

创建程序时，不要对任何一种类型占用的内存量做出假设。

#### signed 和 unsigned

所有整型类型都有两种变体：signed 和 unsigned。有时候，要求整型变量能够存储负数，有时候则不要求。没有使用关键字 unsigned 声明的整型变量都被视为无符号的，这种变量可以为正，也可以为负；而 unsigned 整型变量只能为正。

signed 和 unsigned 整型变量占用的内存空间相同，但 signed 整型变量的部分存储空间被用于存储指出该变量为正还是负的信息，因此 unsigned 整型变量能够存储的最大值为 signed 整型变量能够存储的最大正数的两倍。

例如，如果 short 变量占用 2 字节，则 unsigned short 变量的取值范围为 0~65 535，而 signed short 变量的取值范围内一半为正数，即它能够存储的最大正数为 32 767。然而，signed short 变量也能够存储负数，因此其取值范围为 -32 768~32 767。

### 3.1.4 基本变量类型

C++ 内置了几种变量类型，它们分为整型变量、浮点变量和字符变量。

浮点变量的值可以表示为小数，也就是说它们是实数。字符变量只占 1 字节，通常用于存储 ASCII 字符集和扩展 ASCII 字符集中的 256 个字符和符号。

注意

ASCII 字符集是计算机使用的标准化字符集。ASCII 是 America Standard Code for Information Interchange（美国信息交换标准码）的缩写。几乎所有的计算机操作系统都支持 ASCII 码，虽然很多操作系统还支持其他的国际字符集。

表 3.1 描述了 C++ 程序中使用的变量类型，其中列出了变量类型、占有的内存量以及能够存储的值。变量类型能够存储的值是由其长度决定的，请查看程序清单 3.1 的输出，了解在您的计算机上，变量类型的长度是否与此相同。除非读者的计算机使用的是 64 位处理器，否则变量类型的长度很可能与此相同。

表 3.1 变量类型		
类型	长度	值
bool	1 字节	true/false
unsigned short int	2 字节	0~65 535
short int	2 字节	-32 768~32 767
unsigned long int	4 字节	0~4 294 967 295
long int	4 字节	-2 147 483 648~2 147 483 647
int (16 bit)	2 字节	-32 768~32 767
int (32 bit)	4 字节	-2 147 483 648~2 147 483 647
unsigned int (16 位)	2 字节	0~65 535
unsigned int (32 位)	4 字节	0~4 294 967 295
char	1 字节	256 个字符
float	4 字节	1.2e-38~3.4e38
double	8 字节	2.2e-308~1.8e308

注意

根据您使用的计算机和编译器，变量的长度可能与表 3.1 所示不同。有关各种类型的变量能存储的值，请查阅编译器手册。

### 3.2 定义变量

至此，读者看到过很多被创建和使用的变量，下面介绍如何创建变量。

要创建或定义变量，可声明其类型，加上一个或多个空格，然后指出变量名，再加上一个分号。变量名可以是任何字母组合，但其中不能有空格。x、J23qrsnf 和 myAge 都是合法的变量名。好的变量名指出了其用途，使用好的变量名可使程序流程更容易理解。下面的语句定义了一个名为 myAge 的整型变量：

```
int myAge;
```

注意

声明变量时，将为该变量分配内存。此时相应内存中的值就是该变量的值。稍后将介绍如何给内存赋新值。

一种通用的编程惯例是，应避免使用像 J23qrsnf 这样难懂的变量名，也尽量不要使用单个字母（如 x 或 i）作为变量名。尽可能使用有意义的变量名，如 myAge 或 howMany。当您三周后再次阅读程序时，将很容易知道当时编写代码行时使用的变量名的含义。

请根据前几行代码猜测下面几个程序是做什么的。

## 示例 1:

```
int main()
{
    unsigned short x = 10;
    unsigned short y = 11;

    unsigned short z = x * y;

    return 0;
}
```

## 示例 2:

```
int main()
{
    unsigned short Width = 10;
    unsigned short Length = 11;

    unsigned short Area = Width * Length;

    return 0;
}
```

## 注意

注意到在声明 short 变量 Width 和 Length 时, 就将值 10 和 11 赋给了它们, 这被称为变量初始化, 将在本章后面更详细地讨论。

显然, 第 2 个程序的目的更易猜出。输入较长的变量名带来的不便, 因第二个程序更容易理解和维护得到了补偿。

### 3.2.1 区分大小写

C++区分大小写, 换句话说, 大小写字母是不同的。例如, age、Age 和 AGE 是不同的变量。

## 警告

有些编译器允许关闭大小写区分功能。请不要这样做, 因为这样做, 使用其他编译器时程序将不能通过编译, 同时其他 C++程序员将对您的代码感到非常迷惑。

### 3.2.2 命名规则

关于给变量命名的规则有很多, 采用哪种方法关系不大, 在整个程序中保持一致很重要。命名规则不一致将使其他程序员阅读您的代码时感到迷惑。

很多程序员喜欢全部用小写字母表示变量名。如果变量名由两个单词组成 (如 my car), 有两种流行的表示法: my\_car 和 myCar。后者被称为驼峰式表示法, 因为其中的大写字母很像驼峰。

有些人认为下划线表示法 (如 my\_car) 易读, 但另一些人则尽量避免使用下划线, 因为下划线更难输入。本书使用驼峰式表示法: 变量名中第 2 个及以后的单词的首字母均大写, 如 myCar、theQuickBrownFox 等。

许多高级程序员使用匈牙利表示法。匈牙利表示法的基本思想是, 变量名以一组表示其类型的字符打头。例如, 整型变量可能以小写字母 i 打头, 而 long 变量可能以小写字母 l 打头。有关指示 C++ 中其他结构 (如常量、全局变量、指针等) 的表示法, 将在后面介绍。

## 注意

之所以称为匈牙利表示法, 是因为其发明者微软公司的 Charles Simonyi 是匈牙利人。

微软公司近来已摒弃匈牙利表示法, 其 C#设计建议中强烈建议不要使用匈牙利表示法。有关 C# 的理由也适用于 C++。



### 3.2.3 关键字

有些单词被 C++保留，不能用来作变量名。对 C++编译器来说，这些关键字有特殊含义。关键字包括 if、while、for、main 等。表 3.2 和附录 B 列出了 C++定义的关键字。您的编译器可能还保留了其他单词，有关完整的关键字列表，请参阅编译器手册。

表 3.2 C++关键字

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	
另外，下列单词被保留			
And	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

应该

定义变量时先给出类型，然后是变量名。  
使用有意义的变量名。  
别忘了 C++区分大小写。  
务必了解各种变量在内存中占用的字节数及其能够存储哪些值。

不应该

不要将 C++关键字用作变量名。  
不要对存储变量需要多少字节内存进行假设。  
不要将负数赋给 unsigned 变量。

### 3.3 确定变量类型占用的内存量

程序员不应特定数据类型占用的内存量进行假设。虽然在您的计算机以及大多数计算机中，数据类型 int 可能占用 4 个字节，但明智的做法是不要将此视为理所当然，因为处理器类型和编译器可能影响特定数据类型占用的字节数（但在特定计算机中，这是固定的）。因此，C++提供了运算符 sizeof，可帮助计算特定数据类型在程序运行时占据的空间。请编译并运行程序清单 3.1，它将指出在您的计算机上这些类型的长度。

程序清单 3.1 确定在您的计算机上变量类型的长度

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using std::cout;
6:
7:     cout << "The size of an int is:\t\t"
8:         << sizeof(int) << " bytes.\n";
9:     cout << "The size of a short int is:\t"
10:        << sizeof(short) << " bytes.\n";
11:    cout << "The size of a long int is:\t"
12:        << sizeof(long) << " bytes.\n";
13:    cout << "The size of a char is:\t\t"
14:        << sizeof(char) << " bytes.\n";
15:    cout << "The size of a float is:\t\t"
16:        << sizeof(float) << " bytes.\n";
17:    cout << "The size of a double is:\t"
18:        << sizeof(double) << " bytes.\n";
19:    cout << "The size of a bool is:\t"
20:        << sizeof(bool) << " bytes.\n";
21:
22:    return 0;
23: }
```

#### ▼ 输出:

```
The size of an int is:      4 bytes.
The size of a short int is: 2 bytes.
The size of a long int is:  4 bytes.
The size of a char is:     1 bytes.
The size of a float is:    4 bytes.
The size of a double is:   8 bytes.
The size of a bool is:     1 bytes.
```

#### 注意

在您的计算机上, 显示的字节数可能与上面不同。

读者应该非常熟悉程序清单 3.1 中的大部分内容。其中有些代码行被分成多行, 以适合本书的版面, 例如, 第 7 行和第 8 行实际上应该为一行。编译器忽略空白 (空格、制表符和换行符), 因此可以将这两行看成一行。这也是在大部分代码行末尾需要分号 (;) 的原因。

该程序使用的一项新功能是, 在第 7~20 行使用了运算符 `sizeof`。`sizeof` 的用法类似于函数, 被调用时, 它指出作为参数传递给它的类型的长度。例如, 第 8 行将关键字 `int` 传递给了 `sizeof`。本章后面将指出, `int` 用于描述标准整型变量。在使用奔腾 4 和 Windows XP 的计算机上使用 `sizeof` 时, 它将指出 `int` 的长度为 4 字节, 这正好与 `long int` 的长度相同。

## 3.4 一次创建多个变量

可以在一条语句中创建多个类型相同的变量, 方法是先指出变量类型, 然后指定变量名, 并用逗号将变量名分开。例如:

```
unsigned int myAge, myWeight;    // two unsigned int variables
long int area, width, length;    // three long integers
```

从中可知, 变量 `myAge` 和 `myWeight` 都被声明为 `unsigned int` 变量。第 2 行声明了 3 个名为 `area`、`width` 和 `length` 的 `long` 变量。类型 `long` 被用于所有变量, 因此, 不能在同一条语句中定义不同类型的变量。

## 3.5 给变量赋值

可以使用赋值运算符 (=) 给变量赋值。要将 5 赋给变量 `width`, 可以这样做:

```
unsigned short width;  
width = 5;
```

**注意**

long 是 long int 的简写；short 是 short int 的简写。

可以将创建变量和给它赋值的步骤合而为一。例如，对于变量 width，将这两个步骤合而为一的代码如下：

```
unsigned short width = 5;
```

初始化与前面的赋值极其相似，对于像 width 这样的整型变量，两者的差别很小。后面介绍 const 时，读者将发现，有些变量必须初始化，因为以后不能给它们赋值。

可以一次定义多个变量，也可以在创建时初始化多个变量。例如，下面的代码创建两个 long 变量，并初始化它们：

```
long width = 5, length = 7;
```

上述代码将 long 变量 width 和 length 分别初始化为 5 和 7。您甚至可以将定义和初始化合起来：

```
int myAge = 39, yourAge, hisAge = 40;
```

上述代码创建 3 个 int 变量，并初始化第 1 个 (myAge) 和第 3 个 (hisAge) 变量。

程序清单 3.2 是一个可以编译的完整程序，它计算矩形的面积并将结果打印到屏幕上。

**程序清单 3.2 演示变量的用法**

```
1: // Demonstration of variables  
2: #include <iostream>  
3:  
4: int main()  
5: {  
6:     using std::cout;  
7:     using std::endl;  
8:  
9:     unsigned short int Width = 5, Length;  
10:    Length = 10;  
11:  
12:    // create an unsigned short and initialize with result  
13:    // of multiplying Width by Length  
14:    unsigned short int Area = (Width * Length);  
15:  
16:    cout << "Width: " << Width << endl;  
17:    cout << "Length: " << Length << endl;  
18:    cout << "Area: " << Area << endl;  
19:    return 0;  
20: }
```

**▼ 输出：**

```
Width:5  
Length: 10  
Area: 50
```

**▼ 分析：**

从上述程序清单可知，第 2 行的 include 语句包含 iostream 库，以便能够使用 cout。程序从第 4 行的 main() 函数开始。第 6 行和第 7 行指出 cout 和 endl 位于标准 (std) 名称空间中。

在第 9 行，定义了第一个变量。Width 被定义为 unsigned short 变量，并被初始化为 5。同时，还定义了另一个 unsigned short 变量 Length，但没有初始化。在第 10 行，将值 10 赋给了 Length。

在第 14 行，定义了 unsigned short 变量 Area，并将其初始化为 Width 和 Length 的乘积。在第 16~18 行，将各个变量的值打印到屏幕上。注意，特殊单词 endl 另起一行。

## 3.6 使用 typedef 创建别名

不断输入 unsigned short int 既烦琐又容易出错。C++ 允许您使用关键字 typedef (表示类型定义) 为

这个短语创建一个别名。

实际上，这创建的是一个同义词，将其与第 6 章将介绍的创建新类型区分开来至关重要。要创建别名，可使用关键字 `typedef`，后面跟现有类型和新名称，并以分号结束。例如，下面的语句创建新名称 `USHORT`，您可以在任何本应使用 `unsigned short int` 的地方使用它：

```
typedef unsigned short int USHORT;
```

程序清单 3.3 与程序清单 3.2 相同，但使用了类型定义 `USHORT`，而不是 `unsigned short int`。

程序清单 3.3 演示 `typedef`

```
1: // Demonstrates typedef keyword
2: #include <iostream>
3:
4: typedef unsigned short int USHORT;    //typedef defined
5:
6: int main()
7: {
8:
9:     using std::cout;
10:    using std::endl;
11:
12:    USHORT Width = 5;
13:    USHORT Length;
14:    Length = 10;
15:    USHORT Area = Width * Length;
16:    cout << "Width: " << Width << endl;
17:    cout << "Length: " << Length << endl;
18:    cout << "Area: " << Area << endl;
19:    return 0;
20: }
```

▼ 输出：

```
Width:5
Length: 10
Area: 50
```

注意

星号 (\*) 表示乘法运算。

▼ 分析：

在第 4 行，`USHORT` 被定义为 `unsigned short int` 的同义词。该程序与程序清单 3.2 极其相似，输出也相同。

### 3.7 何时使用 `short` 和 `long`

一个令 C++ 新手感到困惑的问题是，什么时候将变量声明为 `short` 类型，什么时候将其声明为 `long` 类型。规则非常简单：可能需要将比其类型能够存储的最大值还要大的值赋给变量时，应将该变量声明为更长的类型。

如表 3.1 所示，`unsigned short` 变量占用 2 字节，能够存储的最大值为 65 535；`signed short` 变量的取值范围由正数和负数两部分组成，因此能够存储的最大值为前者的一半。

虽然 `unsigned long` 变量可存储非常大的值（4 294 967 295），但这依然很有限。如果需要存储更大的值，必须使用 `float` 或 `double` 变量，但这样会降低精度。`float` 和 `double` 变量能够存储非常大的值，但在大多数计算机上，只有前 7 位或 9 位有效，这意味着其后位将被四舍五入。

变量越短，占用的越少。现在，内存已很便宜，而时间却很宝贵。请放心使用 `int`，在您的计算机上，它可能占用 4 字节。

### 3.7.1 unsigned 整型变量的回绕

unsigned long 变量能够存储的最大值有限，但这通常不是问题。如果数据确实超过了允许的最大值，将出现什么情况呢？

当 unsigned 整型变量达到其最大值时将回绕，就像汽车里程表一样。程序清单 3.4 演示了将过大的值赋给 short 变量时出现的情况。

程序清单 3.4 将过大的值赋给 unsigned short 变量

```
1: #include <iostream>
2: int main()
3: {
4:     using std::cout;
5:     using std::endl;
6:
7:     unsigned short int smallNumber;
8:     smallNumber = 65535;
9:     cout << "small number:" << smallNumber << endl;
10:    smallNumber++;
11:    cout << "small number:" << smallNumber << endl;
12:    smallNumber++;
13:    cout << "small number:" << smallNumber << endl;
14:    return 0;
15: }
```

▼ 输出:

```
small number:65535
small number:0
small number:1
```

▼ 分析:

在第 7 行，smallNumber 被声明为 unsigned short int 变量，在笔者使用的运行 Windows XP 的奔腾 4 计算机上，这种变量占用 2 字节，取值范围为 0~65 535。在第 8 行，将可存储的最大值赋给了 smallNumber，第 9 行打印它。

在第 10 行，smallNumber 被递增，即将其加 1。递增运算符为++，就像 C++表明它是从 C 语言增补而来的。这样，smallNumber 的值将为 65 536，然而，unsigned short 变量不能存储大于 65 535 的值，因此这个值回绕为 0，第 11 行打印了它。

在第 12 行，再次对 smallNumber 进行了递增，然后打印新的值 1。

### 3.7.2 signed 整型变量的回绕

signed 整型变量与 unsigned 整型变量的区别在于，在它能够存储的值中，有一半为负。假设您要模拟的不是传统的汽车里程表，而是如图 3.2 所示的钟表，其中的数字按顺时针递增，按逆时针递减，它们钟面底部（即 6 点钟处）交叉。



图 3.2 如果钟表使用有符号数字



从 0 开始数，下一个数为 1（顺时针方向）或-1（逆时针方向）。数完正数后，接下来就是最大的负数，最终数会到 0。程序清单 3.5 说明了当 short 整型变量为最大允许正数时再加 1 后发生的情况。

程序清单 3.5 将过大的值赋给 signed short 变量

```
1: #include <iostream>
2: int main()
3: {
4:     short int smallNumber;
5:     smallNumber = 32767;
6:     std::cout << "small number:" << smallNumber << std::endl;
7:     smallNumber++;
8:     std::cout << "small number:" << smallNumber << std::endl;
9:     smallNumber++;
10:    std::cout << "small number:" << smallNumber << std::endl;
11:    return 0;
12: }
```

▼ 输出:

```
small number:32767
small number:-32768
small number:-32767
```

▼ 分析:

在第 4 行，smallNumber 被定义为 signed short 变量（如果没有指明为无符号的，整型变量被视为有符号的）。该程序与前一个程序几乎相同，但输出完全不同。要完全理解其输出，必须知道有符号数如何被表示为两字节整型中的位。然而，有符号整型的回绕与无符号整型相同，从最大正值回绕为最小负值。这也称为溢出。

### 3.8 使用字符

字符变量（类型为 char）通常只占 1 字节，能够存储 256 个值（见附录 C）。char 变量可被解释为 0~255 的数或 ASCII 字符集中的成员。ASCII 字符集和对应的 ISO 字符集是一种将字母、数字、标点符号进行编码的方式。

注意

计算机没有字母、标点符号和句子的概念，只能理解数字。实际上，计算机真正能识别的只是线路结点处是否有足够的电压；这两种状态用符号 1 和 0 表示。通过一系列 0 和 1 的组合，计算机能生成可被解释为数字的模式，而数字又可进一步被解释为字母和标点符号。

小写字母 a 的 ASCII 码值为 97。所有大小写字母、数字、标点符号的 ASCII 码值都为 1~128。另外的 128 个标记和符号被留给计算机制造商使用，虽然 IBM 的扩展字符集在某种程度上已成为标准。

#### 3.8.1 字符和数字

将一个字符（如字母 a）赋给 char 变量时，该变量实际存储的是一个 0~255 的值。然而，编译器知道如何在字符（用单引号括起的字母、数字或标点符号）和 ASCII 码值之间进行转换。

这种数值与字符之间的关系是任意的，也就是说，小写字母 a 并非一定要对应 97。只要所有设备（键盘、编译器、显示器）就数值与字符之间的关系达成一致，就不会出现问题。然而，需要指出的是，数值 5 与字符 5 之间有天壤之别。后者对应的 ASCII 码值为 53，就像 a 的 ASCII 码为 97 一样。程序清单 3.6 说明了这一点。

程序清单 3.6 使用数字来打印字符

```
1: #include <iostream>
2: int main()
3: {
4:     for (int i = 32; i<128; i++)
5:         std::cout << (char) i;
6:     return 0;
7: }
```

▼ 输出:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~?
```

▼ 分析:

这个简单程序打印 ASCII 码为 32~127 的字符。它在第 4 行使用一个整型变量 i 来完成这项任务。在第 5 行，强制将变量 i 的值显示为字符。也可以使用字符变量，如程序清单 3.7 所示，其输出相同。

程序清单 3.7 使用数字来打印字符

```
1: #include <iostream>
2: int main()
3: {
4:     for (unsigned char i = 32; i<128; i++)
5:         std::cout << i;
6:     return 0;
7: }
```

正如读者看到的，第 4 行使用了一个 unsigned char 变量。由于使用的是字符变量而不是数值变量，因此第 5 行的 cout 知道应该显示字符值。

### 3.8.2 特殊打印字符

C++编译器能够识别一些特殊的格式化字符。表 3.3 列出了最常用的格式化字符。要在代码中使用它们，可输入反斜杠（被称为转义字符）和相应的字符。例如，要在代码中加入制表符，可依次输入左单引号、反斜杠、字母 t、右单引号：

```
char tabCharacter = '\t';
```

上述代码声明了一个 char 变量（tabCharacter），并将其初始化为t——制表符。特殊打印字符用于将输出发送到屏幕、文件或其他输出设备。

转义字符（\）改变了其后面的字母的含义。例如，通常情况下，n 表示字母 n，但在前面加上转义字符后，表示换行。

表 3.3 转义字符

字符	含义	字符	含义
\a	响铃	\'	单引号
\b	退格	\"	双引号
\f	换页	\?	问号
\n	换行	\\	反斜杠
\r	回车	\000	八进制表示
\t	制表符	\xhhh	十六进制表示
\v	垂直制表符		

## 3.9 常量

与变量一样，常量也是数据的存储位置；与变量不同的是，顾名思义，常量的值不能修改。创建常量时必须初始化，且以后不能给它赋值。

C++有两种常量：字面常量和符号常量。

### 3.9.1 字面常量

字面常量指直接输入到程序中的值。例如，在下面的代码中：

```
int myAge = 39;
```

myAge 是一个 int 变量，而 39 是一个字面常量。不能给 39 赋值，其值也不能修改。

### 3.9.2 符号常量

符号常量指用名称表示的常量，就像表示变量一样。然而，与变量不同的是，常量一旦被初始化，其值就不能改变。

程序中有两个整型变量：students 和 classes，如果每班有 15 个学生，给定班级数，将可以求出学生数：

```
students = classes * 15;
```

在这个例子中，15 是一个字面常量。如果用一个符号常量代替它，程序将更容易理解和维护：

```
students = classes * studentsPerClass
```

如果以后想修改每班的人数，只需在定义常量 studentsPerClass 的地方进行修改，而不必逐个修改。

在 C++ 中，有两种声明符号常量的方法。老式（传统）方法是使用预处理器编译指令 #define，这种方法现已摒弃。另一种也是更合适的方法是，使用关键字 const 来创建它们。

#### 1. 使用 #define 定义常量

由于很多现有的程序使用预处理器编译指令 #define，因此了解其用法很重要。要以这种过时的方式定义常量，可以这样做：

```
#define studentsPerClass 15
```

请注意，这里没有指定 studentsPerClass 的类型（int、char 等）。预处理器进行简单的文本替换。对这个例子而言，每当预处理器看到 studentPerClass 时，都将它替换为文本 15。

由于预处理器在编译器之前运行，因此编译器只能看到 15，而看不到常量 studentsPerClass。

#### 警告

虽然 #define 看似使用起来非常容易，但应避免使用它，因为 C++ 标准指出它已过时。

#### 2. 用 const 定义常量

虽然 #define 可行，但 C++ 中有一种更好的定义常量的方法：

```
const unsigned short int studentsPerClass = 15;
```

这个例子也声明了一个名为 studentsPerClass 的符号常量，但这次 studentsPerClass 被声明为 unsigned short int 类型。

这种声明常量的方法在使程序易于维护和防止错误方面有若干优点。最大的不同在于，该常量有类型，这使得编译器能够根据类型确保它被正确使用。

#### 注意

在程序运行过程中，常量不能被修改。例如，要修改 studentsPerClass 的值，应修改源代码并重新编译。

应该	不应该
谨防整型变量的值超出允许的范围，否则将回绕得到错误的值。 务必使用能反映变量用途的变量名。	不要将 C++ 关键字用作变量名。 不要使用预处理器编译指令 <code>#define</code> 来声明常量，而应使用 <code>const</code> 。

## 3.10 枚举常量

枚举型常量使您能够创建新类型，然后定义新类型变量，将这些变量的取值限定为一组可能值。例如，可以创建一个枚举来存储颜色。具体地说，可以将 `COLOR` 声明为枚举，然而指定 `COLOR` 的 5 个可能取值：`RED`、`BLUE`、`GREEN`、`WHITE` 和 `BLACK`。

创建枚举型常量的语法是：先输入关键字 `enum`，接着为新的类型名、左大括号、用逗号分隔的所有合法值、右大括号和分号，如下例所示：

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

该语句执行两项任务：

- 将 `COLOR` 指定为枚举的名称，即它将是一种新类型；
- 使 `RED` 成为一个符号常量，其值为 0；`BLUE` 也为符号常量，其值为 1；`GREEN` 为符号常量，值为 2；依此类推。

每个枚举型常量都有一个整数值。如不特别指定，第一个常量的值为 0，其余常量依次递增。然而，可将其中任何一个常量初始化为特定的值，这样，后面未被初始化的常量将在这个基础依次递增。因此，如果有下面的代码：

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

则 `RED` 的值将为 100，`BLUE` 的值为 101，`GREEN` 的值为 500，`WHITE` 的值为 501，`BLACK` 的值为 700。

可以定义类型为 `COLOR` 的变量，但它们的值只能是某个枚举值（在这个例子中，为 `RED`、`BLUE`、`GREEN`、`WHITE` 或 `BLACK`）。可以将其中的任何颜色值赋给 `COLOR` 变量。

需要注意的是，枚举变量的类型通常为 `unsigned int`，而枚举常量相当于整型常量。然而，使用颜色、星期几及类似的信息时，如果能够给这些值指定名称，将非常方便。程序清单 3.8 是一个使用枚举类型的程序。

程序清单 3.8 枚举型常量

```
1: #include <iostream>
2: int main()
3: {
4:     enum Days { Sunday, Monday, Tuesday,
5:                 Wednesday, Thursday, Friday, Saturday };
6:
7:     Days today;
8:     today = Monday;
9:
10:    if (today == Sunday || today == Saturday)
11:        std::cout << "\nGotta' love the weekends!\n";
12:    else
13:        std::cout << "\nBack to work.\n";
14:
15:    return 0;
16: }
```

### ▼ 输出：

Back to work.

**▼ 分析:**

在第 4 行和第 5 行, 定义了枚举类型 DAYS, 它有 7 个可能的取值。每个值对应一个整数, 从 0 开始计算, 因此 Monday 的值为 1 (Sunday 的值为 0)。

在第 7 行, 创建了一个类型为 Days 的变量, 也就是说, 这个变量将可以取第 4 行和第 5 行定义的枚举常量之一。第 8 行将值 Monday 赋给该变量; 第 10 行对这个变量的进行检测。

可以使用一系列整型常量而不是枚举类型, 如程序清单 3.9 所示。

**程序清单 3.9 使用整型常量**

```
1: #include <iostream>
2: int main()
3: {
4:     const int Sunday = 0;
5:     const int Monday = 1;
6:     const int Tuesday = 2;
7:     const int Wednesday = 3;
8:     const int Thursday = 4;
9:     const int Friday = 5;
10:    const int Saturday = 6;
11:
12:    int today;
13:    today = Monday;
14:
15:    if (today == Sunday || today == Saturday)
16:        std::cout << "\nGotta' love the weekends!\n";
17:    else
18:        std::cout << "\nBack to work.\n";
19:
20:    return 0;
21: }
```

**▼ 输出:**

Back to work.

**警告**

在这个程序中声明的很多常量没有被使用过, 因此编译该程序清单时, 编译器可能发出警告。

**▼ 分析:**

该程序清单的输出与程序清单 3.8 相同。其中每个常量 (Sunday、Monday 等) 都被显式地定义, 且使用枚举类型 Days。枚举常量的优点一目了然: 枚举类型 Days 的含义是非常清楚的。

## 3.11 总结

本章讨论了数值变量、字符变量和常量, C++使用它们在程序执行期间存储数据。数值变量可以是整型 (char、int 和 long int) 或浮点型 (float、double 和 long double)。另外, 数值变量还可以是有符号或无符号的。虽然各种变量类型的长度随计算机而异, 但在给定的计算机上, 类型决定了变量的长度。

使用变量前必须声明它, 然后只能将声明类型的数据存储到该变量中。如果将过大的值赋给整型变量, 将回绕, 导致错误的结果。

本章还介绍了字面常量、符号常量和枚举常量。读者学习了两种声明符号常量的方法: 使用#define 和关键字 const, 但后者是一种更合适的方式。



## 3.12 问与答

问：既然 short int 可能回绕，为什么不总是使用 long 变量呢？

答：所有整型变量都可能由于空间不够而回绕，但对于 long 变量，需要将更大的值赋给它时才会回绕。例如，两字节的 unsigned short int 变量在赋给它的值大于 65 535 时回绕，而 4 字节的 unsigned short int 变量在赋给它的值大于 4 294 967 295 时回绕。然而，在大多数计算机上，long 变量占用的内存是 int 变量的 2 倍（即 4 字节而不是 2 字节），如果程序中有 100 个这样的变量，将多消耗 200 字节的 RAM。坦率地说，与过去相比，这已经不是什么大问题，因为现在的个人计算机即使没有数千兆字节也有数兆字节内存。

问：如果将一个带小数点的数赋给整型变量而不是浮点变量，将出现什么结果？如下述代码所示：

```
int aNumber = 5.4;
```

答：好的编译器会发出警告，但这种赋值是完全合法的。这个值将被截短为整数。因此，将 5.4 赋给一个整型变量，该变量的值将为 5。然而，将丢失信息，如果再将这个变量的值赋给一个 float 变量，后者的值也将为 5。

问：为什么不用字面常量，为什么要自找麻烦地使用符号常量？

答：如果在程序的很多地方都使用了同一个值，则使用符号常量时，只需修改该常量的定义，就可以修改所有地方的这个值。另外，符号常量的含义明确，在程序中乘以 360 时不好理解，但乘以符号常量 degreesInACircle 将一目了然。

问：如果将负数赋给一个 unsigned 变量将出现什么结果？如下述代码行所示：

```
unsigned int aPositiveNumber = -1;
```

答：好的编译器会发出警告，但这种赋值是合法的。负数将被解释为位模式，并赋给变量。然后该变量的值将被解释为无符号数。-1 的位模式为 1111 1111 1111 1111（用十六进制表示为 0xFF），它将被解释为无符号值 65 535。

问：如果不懂位模式、二进制算术和十六进制，还能使用 C++吗？

答：能，但不如了解这些知识的好。在无需了解计算机的工作原理方面，C++做得没有有些语言那么好，但这实际上是件好事，因为它提供了大量其他语言没有的功能。和任何功能强大的工具一样，要充分发挥 C++的潜力，必须了解其工作原理。使用 C++编程的程序员如果不了解二进制，将经常对结果感到迷惑。

## 3.13 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 3.13.1 测验

1. 整型变量和浮点变量有何不同？
2. unsigned short int 变量和 long int 变量有何不同？
3. 符号常量相对于字面常量有何优点？

4. 使用关键字 `const` 与使用 `#define` 相比有何优点?

5. 什么决定了变量名是好还是坏?

6. 给定如下枚举类型, `BLUE` 的值是多少?

```
enum COLOR { WHITE, BLACK = 100, RED, BLUE, GREEN = 300 };
```

7. 下列变量名哪些是好的? 哪些是不好的? 哪些是非法的?

- a. `Age`
- b. `!ex`
- c. `R79J`
- d. `TotalIncome`
- e. `__Invalid`

### 3.13.2 练习

1. 要存储下列信息, 正确的变量类型是什么?

- a. 年龄。
- b. 后院的面积。
- c. 银河系中的星星数目。
- d. 一月份的平均降水量。

2. 给上述信息取一个好的变量名。

3. 声明一个常量来表示 `pi` 等于 3.141 59。

4. 声明一个 `float` 变量, 并用常量 `pi` 来初始化它。

## 第 4 章

# 管理数组和字符串

在前几章中，读者声明单个 `int` 变量、`char` 变量或其他对象。经常需要声明一组对象，如 20 个 `int` 变量或一组 `Cat` 对象。

在本章中，您将学习：

- 什么是数组以及如何声明它们
- 什么是字符串以及如何使用字符数组表示字符串

### 4.1 什么是数组

数组是一个顺序数据存储单元集合，其中每个存储单元存储相同类型的数据。存储单元被称为数组元素。

要声明数组，可指定类型数组名和下标。下标用方括号括起，指定了数组包含多少个元素。例如：

```
long LongArray[25];
```

声明了一个名为 `LongArray` 的数组，它包含 25 个 `long` 元素。编译器看到该声明时，分配足够的内存来存储这 25 个元素。由于每个 `long` 变量占用 4 字节，因此该声明分配 100 字节的连续内存，如图 4.1 所示。

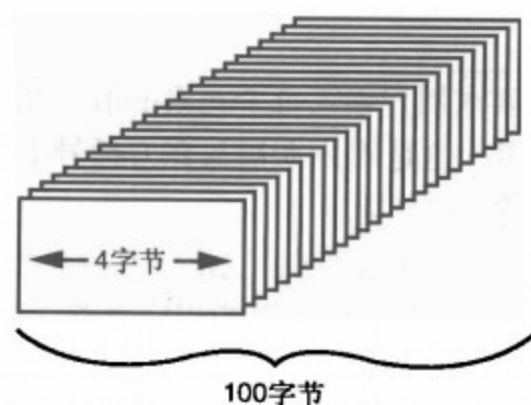


图 4.1 声明数组

#### 4.1.1 访问数组元素

通过使用相对于数组开头的偏移量，可以访问数组元素。数组元素偏移量从 0 开始。因此第一个元素为 `arrayName[0]`。在 `LongArray` 示例中，`LongArray[0]` 是第一个数组元素，`LongArray[1]` 是第二个，依次类推。

这可能有些令人迷惑。数组 `SomeArray[3]` 有三个元素，它们分别是 `SomeArray[0]`、`SomeArray[1]` 和 `SomeArray[2]`。推而广之，`SomeArray[n]` 有 `n` 个元素，分别是 `SomeArray[0]` 到 `SomeArray[n-1]`。这是因为索引是偏移量，因此第一个元素位于从数组开头的第 0 个存储单元中，第二个元素位于第一个存储单元中，依此类推。

因此，LongArray[25]的元素为 LongArray[0]到 LongArray[24]。程序清单 4.1 演示了如何声明一个包含 5 个元素的 int 数组并给每个元素赋值。

注意

从本章开始，程序清单中的行号将从零开始，旨在帮助读者牢记 C++ 中的数组索引从零开始。

程序清单 4.1 使用 int 数组

```
0: //Listing 4.1 - Arrays
1: #include <iostream>
2:
3: int main()
4: {
5:     int myArray[5];    // Array of 5 integers
6:     int i;
7:     for ( i=0; i<5; i++) // 0-4
8:     {
9:         std::cout << "Value for myArray[" << i << "]: ";
10:        std::cin >> myArray[i];
11:    }
12:    for (i = 0; i<5; i++)
13:        std::cout << i << ": " << myArray[i] << std::endl;
14:    return 0;
15: }
```

▼ 输出:

```
Value for myArray[0]: 3
Value for myArray[1]: 6
Value for myArray[2]: 9
Value for myArray[3]: 12
Value for myArray[4]: 15
0: 3
1: 6
2: 9
3: 12
4: 15
```

▼ 分析:

程序清单 4.1 创建一个数组，要求用户输入每个元素的值，然后将这些值打印到控制台。第 5 行声明了一个名为 myArray 的 int 数组，这是因为 5 被放在方括号中，这意味着 myArray 能够存储 5 个整数，其中每个元素都可被视为一个 int 变量。

从第 7 行开始是一个 for 循环，它从 0 数到 4，这正好对应于包含 5 个元素的数组的索引。第 9 行提示用户输入一个值，第 10 行将这个值存储到数组的相应位置。

从第 10 行可知，每个元素都是使用数组名和用方括号括起的偏移量来访问的。然后，每个元素都可被视为一个变量，其类型为数组的类型。

第一个值存储在 myArray[0]中，第二个存储在 myArray[1]中，依次类推。在第 12 行和第 13 行，第二个 for 循环将每个值显示到屏幕上。

注意

数组索引从 0 而不是 1 开始。在 C++ 新手编写的程序中，这是导致很多错误的原因。将索引视为偏移量。第一个元素（如 ArrayName[0]）位于数组开头，因此偏移量为 0。因此，使用数组时，别忘了包含 10 个元素的数组的元素为 ArrayName[0]到 ArrayName[9]，使用元素 ArrayName[10]是错误的。

### 4.1.2 在数组末尾后写入数据

将值写入到数组元素中时，编译器根据每个元素的大小和下标来确定将这个值存储到哪里。假设

您要求将一个值写入 `LongArray[5]` (第 6 个元素) 中, 编译器将偏移量 (5) 和元素的大小 (这里为 4) 相乘。然后从数组开头向前移动相应数量 (20) 的字节, 并将新值写入到这个位置。

如果您将输入写入到 `LongArray[50]` 中, 大多数编译器对“没有这样的元素”视而不见, 而是计算从第一个元素开始需要移动多少个字节 (200), 然后将值写入到这个位置。这里存储的可能是任何数据, 将新值写入到这里可能导致不可预料的结果。如果幸运的话, 程序将立即崩溃; 如果不那么幸运, 在很久以后程序将产生奇怪的结果, 且很难确定程序在什么地方出了问题。

编译器就像一个以步数测量距离的盲人。他从第一座房子 `MainStreet[0]` 开始, 当您要求他前往 `Main Street` 的第 6 座房子时, 他自言自语地说: 我必须再穿过 5 座房子, 每座房子为 4 大步, 因此还要走 20 步。如果您要他去 `MainStreet[100]`, 而 `Main Street` 上只有 25 栋房子, 他将向前走 400 步。远在到达目的地之前, 他肯定地会撞向一辆卡车。因此, 要他去什么地方之前, 您一定要三思。

程序清单 4.2 在数据末尾后面写入数据。读者应编译该程序, 看是否出现错误和警告消息。如果没有, 在使用数组时一定要多加小心!

**注意**

千万别运行该程序, 它可能导致系统崩溃。

**程序清单 4.2 在数组末尾后写入数据**

```
0: Listing 4.2 demonstrates what writing past the bounds of an array is!
1:
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     long TargetArray[25]; // array to fill
8:
9:     int i;
10:    for (i=0; i<25; i++)
11:        TargetArray[i] = 10;
12:
13:    cout << "Test 1: \n"; // test current values (should be 0)
14:    cout << "TargetArray[0]: " << TargetArray[0] << endl; // lower
    ➔bound
15:    cout<<"TargetArray[24]: "<<TargetArray[24]<<endl<<endl; // upper bound
16:
17:    cout << "\nAttempting at assigning values beyond the upper-bound...";
18:    for (i = 0; i<=25; i++) // Going a little too far!
19:        TargetArray[i] = 20; // Assignment may fail for element [25]
20:
21:    cout << "\nTest 2: \n";
22:    cout << "TargetArray[0]: " << TargetArray[0] << endl;
23:    cout << "TargetArray[24]: " << TargetArray[24] << endl;
24:    cout<<"TargetArray[25]: "<<TargetArray[25]<<endl<<endl; // out of bounds^
25:
26:    return 0;
27: }
```

**▼ 输出:**

```
<Program may terminate in a stack-corruption warning>
Test 1:
TargetArray[0]: 10
TargetArray[24]: 10

Attempting at assigning values beyond the upper-bound...
Test 2:
TargetArray[0]: 20
TargetArray[24]: 20
TargetArray[25]: 20
```



## ▼ 分析:

这个简单的应用程序执行的操作非常危险。正如读者看到的,对 long 数组 TargetArray 赋值了两次,但第二次赋值时(如第 18 行所示),for 循环跨越数组边界给 TargetArray[25]赋值。在 C++中,数组索引从 0 开始,因此包含 25 个元素的数组 TargetArray 的索引范围为 0~24。所以给 TargetArray[25]赋值是非法的,同样第 24 行读取 TargetArray[25]的值也是非法的。这称为“缓冲区溢出”错误,且不能保证该程序能够正确运行(即使它在开发环境中能够正确运行)。

## 注意

由于编译器使用内存的方式不同,因此读者的结果可能与此不同。无论如何,读写超越数组边界的元素都将导致“存取非法”,应不惜一切代价避免这种情况发生。

## 4.1.3 护栏柱错误

由于越过数组末尾进行写操作的错误很常见,因此它有专用名称:护栏柱错误(fence post error)。它指的是这样一种问题:要建立一堵长 10 英尺(ft)的护栏,如果两个相邻护栏柱相隔 1 英尺,需要多少个护栏柱?大多数人会说 10 个,但实际上需要 11 个。图 4.2 说明了这一点。

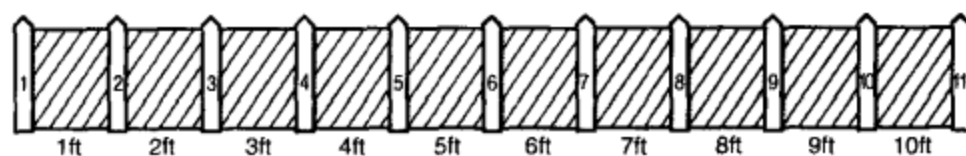


图 4.2 护栏柱错误

这种“少 1”(off by one)计数可能是 C++ 程序员的恶梦。然而,随着时间的推移,读者将习惯这样一种观念:包含 25 个元素的数组的最大索引为 24,一切都是从 0 开始计数的。

## 注意

有些程序员将 ArrayName[0]称为第 0 个元素。养成这种习惯是种错误。如果 ArrayName[0]是第 0 个元素,ArrayName[1]又是什么呢?第一个元素吗?如果是的话,那么当您看到 ArrayName[24]时,认为它不是第 24 个而是第 25 个元素吗?更不会令人迷惑的说法是: ArrayName[0]的偏移量为零,是第一个元素。

## 4.1.4 初始化数组

可以在声明内置类型(如 int 和 char)的简单数组时对其进行初始化。为此,在数组名后加上等号(=)以及一组用大括号括起、用逗号分隔的值。例如,下面的语句声明了一个包含 5 个元素的 int 数组,并将所有元素的值都初始化为 0:

```
// An array of 5 integers, all elements initialized to 0
```

```
int IntegerArray[5] = {0};
```

下面是将各个元素初始化为 0 的另一种方式:

```
// An array of 5 integers initialized to zero
```

```
int IntegerArray[5] = { 0, 0, 0, 0, 0 };
```

这行代码与前面的代码等效,差别在于它显式地将每个元素初始化为 0。因此,如果需要将数组中的元素初始化为不同的值,可以像下面这样做:

```
// An array of 5 integers with 5 different initial values
```

```
int IntegerArray[5] = { 10, 20, 30, 40, 50 };
```

上述代码将第一个元素 IntegerArray[0]初始化为 10,将元素 IntegerArray[1]初始化为 20,等等。

如果省略数组大小,将创建一个刚好能够存储初始值的数组。因此,如果这样编写代码:

```
// The size is not specified, yet 5 initialization values
// tell the compiler to create an array of 5 integers
```

int IntegerArray[] = { 10, 20, 30, 40, 50 };  
创建的数组将与前一个例子中相同：一个包含 5 个元素的数组。

初始化的元素数不能超过为数组声明的元素数。因此，下面的代码将导致编译错误，因为您声明了一个包含 5 个元素的数组，却提供了 6 个初始值：

```
// Array of 5 integers get assigned 6 initial values --> error!
int IntegerArray[5] = { 10, 20, 30, 40, 50, 60};
```

然而，下面的写法是合法的：  
int IntegerArray[5] = {10, 20};

在这个例子中，您声明了一个包含 5 个元素的数组，但只初始化前两个元素：IntegerArray[0]和 IntegerArray[1]。

应该	不应该
务必初始化数组，否则其元素将包含未知值。 请牢记数组第一个元素的偏移量为 0。	不能在超过数组末尾的地方执行写入操作。 不要使用怪异的数组名，数组名应该向其他变量名一样有意义。

4.1.5 声明数组

数组可以使用任意合法的变量名，但不能与其作用域中的其他变量或数组同名。因此不能同时有一个名为 myCats[5]的数组和一个名为 myCats 的变量。

程序清单 4.2 中的代码使用“魔术数字”25 指定数组 TargetArray 的长度。一种更安全的做法是使用常量指定数组长度，这样只需修改一个地方就可修改所有值。在程序清单 4.2 中，使用的是字面量。如果要修改 TargetArray 使其存储 20 而不是 25 个元素，必须修改多行代码。如果使用常量来指定数组大小，则只需要修改该常量的值。

使用枚举量来指定元素数（数组长度）稍微有些不同，程序清单 4.3 说明了这一点，它创建一个数组来存储一周中每天的值。

程序清单 4.3 使用枚举量来指定数组大小

```
0: // Listing 4.3
1: // Dimensioning arrays with consts and enumerations
2:
3: #include <iostream>
4: int main()
5: {
6:     enum WeekDays { Sun, Mon, Tue,
7:                     Wed, Thu, Fri, Sat, DaysInWeek };
8:     int ArrayWeek[DaysInWeek] = { 10, 20, 30, 40, 50, 60, 70 };
9:
10:    std::cout << "The value at Tuesday is: " << ArrayWeek[Tue];
11:    return 0;
12: }
```

▼ 输出:

The value at Tuesday is: 30

▼ 分析:

第 6 行创建了一个名为 WeekDays 的枚举类型，它有 8 个成员。Sun 的值为 0，DaysInweek 的值为

7. 第8行声明了一个名为 `ArrayWeek` 的数组，它包含 `DaysInWeek` (7) 个元素。

第10行将枚举常量 `Tue` 用作数组索引。由于 `Tue` 的值为2，因此第10行返回并打印数组的第三个元素：`ArrayWeek[2]`。

## 数组

要声明数组，可指定其存储的对象的类型以及数组名和决定数组能存储多少个对象的下标。

示例 1:

```
int MyIntegerArray[90];
```

示例 2:

```
long * ArrayOfPointersToLongs[100];
```

要访问数组成员，可使用下标运算符。

示例 1:

```
// assign ninth member of MyIntegerArray to theNinthInteger
int theNinthInteger = MyIntegerArray[8];
```

示例 2:

```
// assign ninth member of ArrayOfPointersToLongs to pLong.
long * pLong = ArrayOfPointersToLongs[8];
```

数组索引从0开始。包含  $n$  个元素的数组的索引为0到  $n-1$ 。

## 4.2 使用多维数组

到目前为止，读者看到的数组都是一维的，可以像谈论字符串（字符串也是一维的）长度一样谈论数组长度，且可使用单个下标参数指定数组元素，就像可以使用离字符串开头的距离指定字符串中的点一样。

然而，数组可以是多维的。也就是说，就像字符串中的点可以用单维数组表示一样，矩形区域中的点可以用一个多维数组表示。以数组方式表示矩形区域中的点时，需要两个下标参数才能指定区域中的一个点： $x$  和  $y$ （二维）。可以创建包含任意维数的数组，这使得从数学点的角度考虑多维数组很有趣。

### 4.2.1 声明多维数组

数组中的每维用一个下标表示，因此二维数组有两个下标，三维数组有三个下标，依次类推。数组可以有任意维数。

一个典型的二维数组的是棋盘，其中一维代表8行，另一维代表8列，如图4.3所示。

假设棋盘上的每格都用两个整数表示，其中一个为  $x$  位置（水平方向），另一个为  $y$  位置（垂直方向），则声明一个表示棋盘中方格的代码如下：

```
int Board[8][8];
```

也可用一个包含64个元素的一维数组表示棋盘（即所有方格都水平排列），例如：

```
int Board[64];
```

然而，这不如二维数组那样更准确地对应现实世界中的物体。国际象棋开局时“王”位于第1行的第4个位置。假设数组的第一个下标代表行，第二个下标代表列，则这个位置对应于：

```
Board[0][3];
```

### 4.2.2 初始化多维数组

可以初始化多维数组。指定的初始值按如下顺序被赋给数组元素：最后（最右边）的数组下标从0递增，其他下标保持不变。因此，如果有下面这样的数组：

```
int theArray[5][3];
```

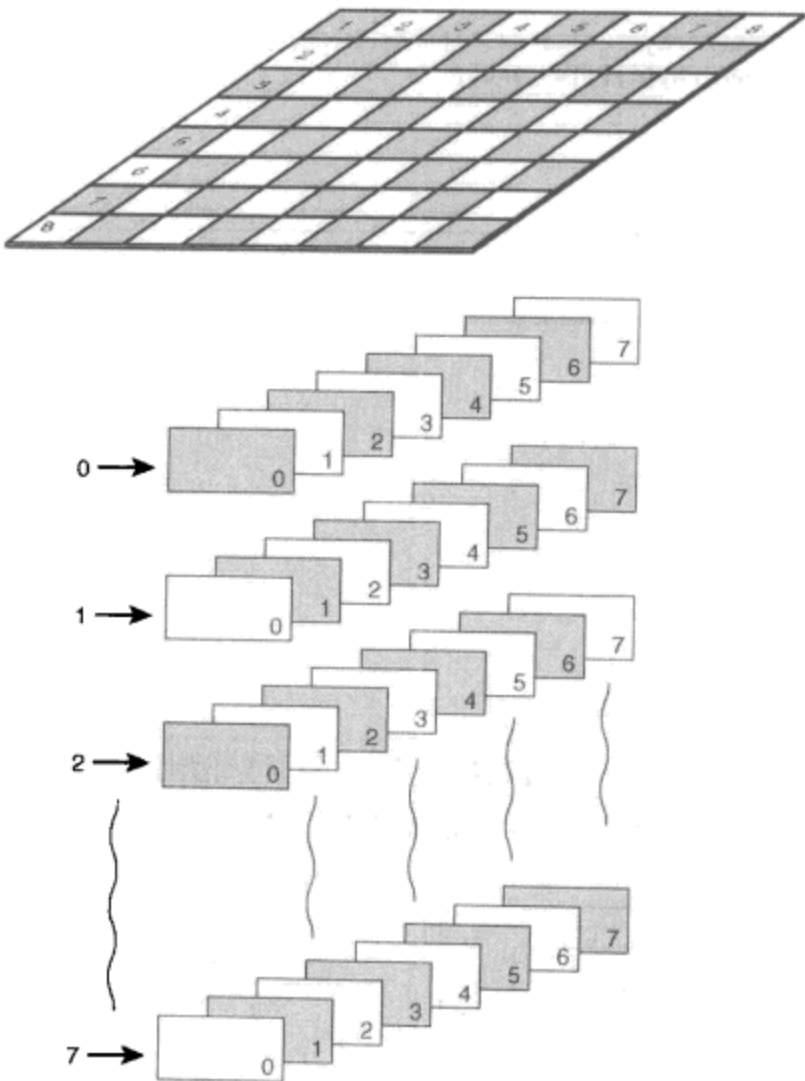


图 4.3 棋盘和二维数组

则前 3 个初始值将存储到 theArray[0]中；接下来的 3 个值存储到 theArray[1]中，依次类推。可以这样来初始化该数组：

```
int theArray[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
出于清晰考虑，可以用大括号将初始值分组，例如：
int theArray[5][3] = { {1,2,3},
    {4,5,6},
    {7,8,9},
    {10,11,12},
    {13,14,15} };
```

编译器将忽略里面的大括号，但它们确实使得更容易理解这些数字是如何分配的。初始化数组元素时，不管是否使用大括号，每个数都必须用逗号分开。全部初始值必须用大括号括起，并以分号结尾。将所有元素都初始化为 0 更简单，如下所示：

```
int theArray[5][3] = {0};
```

编译器将 0 赋给每个元素，从位置[0,0]开始，到位置[4,2]结束。注意，元素位置或索引是从 0 开始的，而在数组定义中指定的是数组包含的元素数。

程序清单 4.4 创建了一个二维数组。第一维由 0~4 组成，第二维元素是第一维对应元素的两倍。

程序清单 4.4 创建多维数组

```
0: // Listing 4.4 - Creating a Multidimensional Array
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     int SomeArray[2][5] = { {0,1,2,3,4}, {0,2,4,6,8}};
7:     for (int i = 0; i<2; i++)
8:     {
9:         for (int j=0; j<5; j++)
```



```
10:    {  
11:        cout << "SomeArray[" << i << "][" << j << "]: ";  
12:        cout << SomeArray[i][j] << endl;  
13:    }  
14: }  
15: return 0;  
16: }
```

#### ▼ 输出:

```
SomeArray[0][0]: 0  
SomeArray[0][1]: 1  
SomeArray[0][2]: 2  
SomeArray[0][3]: 3  
SomeArray[0][4]: 4  
SomeArray[1][0]: 0  
SomeArray[1][1]: 2  
SomeArray[1][2]: 4  
SomeArray[1][3]: 6  
SomeArray[1][4]: 8
```

#### ▼ 分析:

第 6 行将 `SomeArray` 声明为一个二维数组。第一个下标指出有两组，第二个下标指出每组包含 5 个 `int` 值。这相当于创建了一个  $2 \times 5$  的网格，如图 4.4 所示。

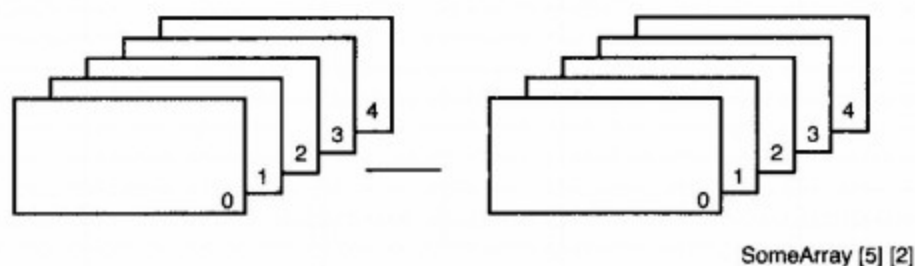


图 4.4 一个  $2 \times 5$  数组

初始值被分成两组，第一组为  $0 \sim 4$ ，第二组为第一组的两倍。在这个程序清单中，直接设置数组元素的值，虽然也可以通过计算得到。第 7 行和第 9 行创建了一个嵌套 `for` 循环。外层 `for` 循环（从第 7 行开始）遍历第一个下标，对于其每个可能值，内层 `for` 循环（从第 9 行开始）遍历第二个下标，这与输出结果是一致的。`SomeArray[0][0]` 的后面是 `SomeArray[0][1]`。仅当遍历完第二个下标的所有可能值后，才将第一个下标加 1，然后重新开始遍历第二个下标的所有可能值。

#### 有关内存的说明

声明数组时，您告诉编译器，要在数组中存储多少个对象。编译器将为所有对象分配内存，即使从不使用它。在知道数组需要存储多少个对象时，这不是问题。例如，棋盘有 64 个方格，而猫产 1~10 个小猫。然而，在不知道需要多少个对象时，必须使用更高级的数据结构。

第 18 章将讨论动态数组 `std::vector` 和 `std::deque`，这些实体的行为是由 C++ 标准委员会指定的。它们向程序员提供了动态数组的泛型实现，可满足大部分需求并解决大部分地址问题，如内存管理、复制等。

## 4.3 字符数组和字符串

有一种数组需要特别注意，这就是以空字符结尾的字符数组。这种数组被称为“C 风格字符串”。到目前为止，读者见到的唯一的 C 风格字符串是在 `cout` 语句中使用的未命名的 C 风格字符串常量，例如：

```
cout << "hello world";
```

可以像其他数组那样声明和初始化 C 风格字符串，例如：

```
char Greeting[] =  
{ 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0' };
```



在这个例子中，Greeting 被声明为一个字符数组，并使用一系列字符对其进行了初始化。最后一个字符'\0'是空字符，很多 C++ 函数都将其视为 C 风格字符串的结束标记。虽然这种逐个字符初始化的方法可行，但输入时太麻烦且容易出错。C++ 提供了一种简便的方法来代替前面的代码，这就是：

```
char Greeting[] = "Hello World";
```

这种语法有两点需要注意：

- 使用双引号将 C 风格字符串括起，没有逗号和大括号；而不是用由逗号将用单引号括起的字符分开，并用大括号将它们括起；
- 不必添加空字符，因为编译器会自动添加。

声明字符串时，应确保其大小能够满足需要。C 风格字符串的长度包括末尾的空字符在内。例如，字符串“Hello World”为 12 个字节，其中 Hello 为 5 字节，空格为 1 字节，World 为 5 字节，空字符为 1 字节。

也可以创建没有被初始化的字符数组。像所有数组一样，必须确保存入的数组不超过其空间。程序清单 4.5 演示了如何使用未初始化的字符数组。

#### 程序清单 4.5 填充数组

```
0: //Listing 4.5 char array buffers
1:
2: #include <iostream>
3:
4: int main()
5: {
6:     char buffer[80] = {'\0'};
7:     std::cout << "Enter the string: ";
8:     std::cin >> buffer;
9:     std::cout << "Here's the buffer: " << buffer << std::endl;
10:    return 0;
11: }
```

#### ▼ 输出：

```
Enter the string: Hello World
Here's the buffer: Hello
```

#### ▼ 分析：

第 6 行声明了一个字符数组，以充当一个能存储 80 个字符的缓冲区，其所有元素都被初始化为空字符\0。该数组最多可存储一个包含 79 个字符的 C 风格字符串和一个结束的空字符。

第 7 行提示用户输入一个 C 风格字符串，第 8 行将其存储到缓冲区中。Cin 将字符串写入缓冲区后加上一个结束的空字符。

程序清单 4.5 中的程序存在两个问题。首先，如果用户输入的字符多于 79 个，cin 将在超出缓冲区末尾的地方写入；其次，如果用户输入了空格，cin 将认为这是字符串的结尾，从而停止向缓冲区写入。

为解决这些问题，必须对 cin 调用一个特殊的方法 get()。cin.get() 接受 3 个参数：

- 待填充的缓冲区；
- 要读取的最大字符数；
- 终止输入的限定符。

默认限定符为换行符。程序清单 4.6 演示了 get() 的用法。

#### 程序清单 4.6 填充数组

```
0: //Listing 4.6 using cin.get()
1:
2: #include <iostream>
3: using namespace std;
4:
5: int main()
```

```

6: {
7:     char buffer[80] = {'\0'};
8:     cout << "Enter the string: ";
9:     cin.get(buffer, 79);        // get up to 79 or newline
10:    cout << "Here's the buffer: " << buffer << endl;
11:    return 0;
12: }

```

**▼ 输出:**

```

Enter the string: Hello World
Here's the buffer: Hello World

```

**▼ 分析:**

第9行调用 `cin` 的 `get()` 方法。第7行声明的缓冲区被作为第一个参数传递给它，第二个参数是要获取的最大字符数。在这个例子中，它不能超过79，这样才有空间存储结尾的空字符。由于默认值换行符足够了，因此不必提供结束字符。

如果用户输入了空格、制表符或其他空白字符，它们将被赋给字符串。换行符结束输入。输入79个字符后，也将结束输入。读者可以再次运行该程序清单，并输入一个超过79个字符的字符串来验证这一点。

## 4.4 使用方法 `strcpy()` 和 `strncpy()`

C++库中有很多可用于处理C风格字符串的函数，其中的很多是从C语言那里继承而来的。在提供的很多函数中，有两个用于将一个字符串复制到另一个字符串中：`strcpy()`和`strncpy()`。`strcpy()`将整个字符串复制到指定的缓冲区中；`strncpy()`将指定数目的字符从一个字符串复制到另一个字符串中。程序清单4.7演示了`strcpy()`的用法。

### 程序清单 4.7 使用 `strcpy()`

```

0: //Listing 4.7 Using strcpy()
1:
2: #include <iostream>
3: #include <string.h>
4: using namespace std;
5:
6: int main()
7: {
8:     char String1[] = "No man is an island";
9:     char String2[80] = {'\0'};
10:
11:     strcpy(String2, String1);
12:
13:     cout << "String1: " << String1 << endl;
14:     cout << "String2: " << String2 << endl;
15:     return 0;
16: }

```

**▼ 输出:**

```

String1: No man is an island
String2: No man is an island

```

**▼ 分析:**

这个程序清单比较简单，它将一个字符串中的数据复制到另一个字符串中。第3行包含了头文件 `string.h`，该文件包含函数 `strcpy()` 的原型。函数 `strcpy()` 接受两个字符数组，前者为源数组，后者为目标数组。第11行使用该函数将 `String1` 复制到 `String2` 中。

使用函数 `strcpy()` 时一定要小心。如果源数组比目标数组长，`strcpy()` 将覆盖缓冲区后面的内容。

为防止这种情况发生，标准库还提供了 `strncpy()`。该变体接受最多要复制的字符数作为参数。函数 `strncpy()` 将第一个空字符前的内容或指定的最大字符数复制到目标缓冲区中。程序清单 4.8 演示了 `strncpy()` 的用法。

程序清单 4.8 使用 `strncpy()`

```
0: //Listing 4.8 Using strncpy()
1:
2: #include <iostream>
3: #include <string.h>
4:
5: int main()
6: {
7:     const int MaxLength = 80;
8:     char String1[] = "No man is an island";
9:     char String2[MaxLength+1] = {'\0'};
10:
11:     strncpy(String2, String1, MaxLength);    // safer than strcpy
12:
13:     std::cout << "String1: " << String1 << std::endl;
14:     std::cout << "String2: " << String2 << std::endl;
15:     return 0;
16: }
```

#### ▼ 输出:

```
String1: No man is an island
String2: No man is an island
```

#### ▼ 分析:

同样，这个程序清单也很简单。和前一个程序清单一样，这里也只是将数据从一个字符串复制到另一个字符串中。在第 11 行调用了 `strncpy()` 而不是 `strcpy()`，它接受第 3 个参数：要复制的最大字符数。缓冲区 `String2` 被声明为能存储 `MaxLength+1` 个字符，多出一个字符用于存储空字符，`strcpy()` 和 `strncpy()` 都会在字符串末尾自动添加一个空字符。

#### 注意

和程序清单 4.9 中的 `int` 数组一样，也可以采用堆分配技术和逐元素复制的方法来调整字符数组的大小。提供给 C++ 程序员的最灵活的 `string` 类使用这种技术的变体来支持增大/缩小字符串以及在字符串中间插入或删除元素。

## 4.5 string 类

C++ 继承了 C 语言中以空字符结尾的 C 风格字符串以及包括 `strcpy()` 函数的函数库。但这些函数并没有集成到面向对象的框架中。像所有数组一样，字符数组也是静态的。定义它们的大小后，即便根本不使用，它们也总是占据相应数量的内存。在超出数组末尾的位置执行写操作将导致灾难性后果。

从前面的示例代码可知，使用诸如 `strcpy()` 等 C 风格函数时，程序员必须负责管理内存。例如，使用 `strcpy` 前，必须确保给目标缓冲区分配了足够的容量，能够存储将复制到其中的字符串，否则写入时将跨越缓冲区边界，导致严重的问题。存储长度可变的用户输入时，这种限制带来了严重的缺点。程序员要么动态地分配目标缓冲区（这要求程序员确定源缓冲区的长度，并正确地分配目标缓冲区），要么使用静态分配的缓冲区（如数组），但其长度是大概估计的，在运行阶段可能发现仍太短。这导致原本非常简单的操作（如复制字符串）充满危险，甚至可能导致应用程序崩溃。

为满足这些经常遇到的需求，C++ 标准库提供了一个 `string` 类，它提供了封装的数据集和操纵这些

字符串数据的函数，使得处理字符串更轻松。std::string 类负责处理内存分配，这使得复制字符串或给它们赋值很轻松，如程序清单 4.9 所示。

程序清单 4.9 使用 std::string 给字符串赋值以及初始化和拼接字符串

```
0: #include <string>
1: #include <iostream>
2:
3: int main ()
4: {
5:     // A sample string
6:     std::string str1 ("This is a C++ string! ");
7:
8:     // display on the console / screen
9:     std::cout << "str1 = " << str1 << std::endl;
10:
11:    // a second sample string
12:    std::string str2;
13:
14:    // assign to make a copy of the first in the second
15:    str2 = str1;
16:
17:    // display the copy
18:    std::cout << "Result of assignment, str2 = " << str2 << std::endl;
19:
20:    // change (overwrite) the second string with a new value
21:    str2 = "Hello string!";
22:
23:    std::cout << "After over-writing contents, str2 = " << str2;
24:    std::cout << std::endl << std::endl;
25:
26:    std::string strAddResult;
27:
28:    // Add the two std::strings (concatenate) and store in a third
29:    strAddResult = str1 + str2;
30:
31:    std::cout << "The result of str1 + str2 is = " << strAddResult;
32:
33:    return 0;
34: }
```

#### ▼ 输出:

```
str1 = This is a C++ string!
Result of assignment, str2 = This is a C++ string!
After over-writing contents, str2 = Hello string!

The result of str1 + str2 is = This is a C++ string! Hello string!
```

#### ▼ 分析:

这里不深入探讨 string 类背后的概念，而简要说明其用法，让您能够在复制字符串时像复制整数一样直观和简单。同样，要拼接两个字符串，只需将其相加，就像将两个整数相加一样。std::string 在幕后负责为程序员管理内存和复制数据，这使得实现快捷、清晰而稳定。这里的一个前提条件是包含头文件<string>，如第 1 行代码所示。

要详细了解 std::string 的各种函数，请参阅第 17 章。由于读者还未学习类和模板，因此现在不要去阅读第 17 章中不熟悉的内容，而将重点放在理解示例的要点上。

## 4.6 总结

本章介绍如何创建数组。数组是大小固定的相同类型的对象集合。

数组不执行边界检查，因此在超出数组末尾的地方进行读写是合法的，虽然其后果是灾难性的。数组索引从 0 开始。一种常见的错误是，将下标 n 用于包含 n 个元素的数组。

数组可以是一维或多维的。无论是一维还是多维的，只要数组包含的元素为内置类型（如 `int`）或有默认构造函数的类，就可以被初始化。

字符串是字符数组。C++ 为管理字符数组提供了专用的功能，其中包括使用用引号的括起字符串来初始化它们。

C++ 标准库还通过头文件 `<string>` 向程序员提供了 `std::string`，这使得复制和操纵字符串很容易。了解如何声明和使用 C 风格字符串后，应使用 C++ 字符串类（如 `std::string`）来编写程序。

## 4.7 问与答

问：未被初始化的数组元素将包含什么内容？

答：分配内存时包含在内存中的内容。使用未初始化的数组元素的结果是不可预测的。如果编译器遵循了 C++ 标准，静态的非局部数组元素将被初始化为零。

问：可以合并数组吗？

答：可以。对于简单数组，可使用指针将它们合并成一个更大的新数组；对于字符串，可使用一些内置函数（如 `strcat`）来合并。

问：诸如 `std::vector` 等动态数组类有何优点？

答：优点在于使用动态数组时，程序员无需在编译阶段知道数组需包含多少个元素。动态数组可根据应用程序的需求动态调整其长度；另外，这些类提供的实用函数对程序员也很有吸引力。

问：字符串类必须使用 `char` 数组来存储字符串的内容吗？

答：不必。可使用设计人员认为的任何最合适的存储方式。

## 4.8 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学的知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

### 4.8.1 测验

1. `SomeArray[25]` 中第一个元素和最后一个元素分别是什么？
2. 如何声明多维数组？
3. 将数组 `SomeArray[2][3][2]` 的所有元素都初始化为 0。
4. 数组 `SomeArray[10][5][20]` 包含多少个元素？
5. 字符串 “Jesse knows C++” 中存储了多少个字符？
6. 字符串 “Brad is a nice guy” 的最后一个字符是什么？

### 4.8.2 练习

1. 声明一个二维数组来表示井字游戏棋盘。
2. 编写将练习 1 中声明的数组中所有元素都初始化为 0 的代码。



3. 编写一个使用了4个数组的程序，其中的3个数组分别存储了您的姓、中间名和名。使用本章介绍的字符串函数将这些字符串复制到存储全名的第4个数组中。

4. 查错：下面的代码段有什么错误？

```
unsigned short SomeArray[5][4];  
for (int i = 0; i<4; i++)  
    for (int j = 0; j<5; j++)  
        SomeArray[i][j] = i+j;
```

5. 查错：下面的代码段有什么错误？

```
unsigned short SomeArray[5][4];  
for (int i = 0; i<=5; i++)  
    for (int j = 0; j<=4; j++)  
        SomeArray[i][j] = 0;
```



## 第 5 章

# 使用表达式、语句和运算符

从本质上说，程序是一组按顺序执行的命令。程序的强大功能源自它能够根据某个条件为真还是假来执行一组或另一组命令。

在本章中，您将学习：

- 什么是语句
- 什么是语句块
- 什么是表示式
- 如何根据条件执行不同的程序代码
- 什么是真值 (truth)，如何使用它

### 5.1 语句简介

在 C++ 中，语句控制程序的执行顺序、计算表达式的值或什么都不做（空语句）。所有的 C++ 语句都以分号结尾。最常见的语句之一是下面的赋值语句，如下所示：

```
x = a + b;
```

与代数中不同，该语句并不表示  $x$  等于  $a + b$ 。它应该读作：将  $a$  与  $b$  的和赋给  $x$  或将  $x$  的值设置为  $a$  与  $b$  的和。

这条语句完成两项工作：将  $a$  和  $b$  相加，并使用赋值运算符 (=) 将结果赋给  $x$ 。虽然这条语句完成两项工作，但它只是一条语句，因此只有一个分号。

注意

赋值运算符将右边的值赋给左边的变量。

#### 5.1.1 使用空白

空白（制表符、空格和换行符）是不可见的，它们被称为空白字符，因为在白纸上打印它们时，看到的只是纸张的白色。

语句中的空白通常被忽略。例如上面讨论的赋值语句可写成：

```
x=a+b;
```

或

```
x      =a  
+      b      ;
```

虽然后一种写是完全合法的，但这么做太傻。空白可使程序更易阅读和维护，但如果使用不当，也可使程序变得很糟糕。在这方面，C++ 提供了强大功能，但程序员需要有判断力。

5.1.2 语句块和复合语句

在任何可以使用单条语句的地方都可以使用复合语句（也叫语句块）。语句块以左大括号（{）开始，以右大括号（}）结束。

虽然语句块中的每条语句都必须以分号结尾，但语句块本身不需要，如下例所示：

```
{
    temp = a;
    a = b;
    b = temp;
}
```

这个代码块是一条语句，它交换变量 a 和 b 的值。

应该	不应该
语句一定要以分号结尾。 使用空白来提高代码的可读性。	使用左大括号后，别忘了与之匹配的右大括号。

5.2 表达式

在 C++ 中，任何结果为一个值的東西都是表达式。表达式总是返回一个值。语句 3+2; 返回值 5，因此它是表达式。所有表达式都是语句。

读者可能感到惊讶，有些代码也是合格的表达式。下面是 3 个这样的例子：

```
3.2           // returns the value 3.2
PI            // float constant that returns the value 3.14
SecondsPerMinute // int constant that returns 60
```

假设 PI 是一个被初始化为 3.14 的常量，SecondsPerMinute 是一个值为 60 的常量，则上面 3 条语句都是表达式。

下面的表达式要复杂些：

```
x = a + b;
```

它将 a 和 b 相加，将结果赋给 x，并返回所赋的值（x 的值）。因此这条语句也是表达式。

注意，任何表达式都可以放在赋值运算符的右边，这包括上述赋值语句。在 C++ 中，下面的代码是完全合法的：

```
y = x = a + b;
```

这行代码的计算顺序如下：

- 1. 将 a 和 b 相加，
- 2. 将表达式 a+b 的结果赋给 x，
- 3. 将赋值表达式 x=a+b 的结果赋给 y。

如果 a、b、x、y 均为整型变量，且 a 的值为 9，b 的值为 7，则 x 和 y 的值都将为 16。程序清单 5.1 说明了这一点。

程序清单 5.1 计算复杂表达式的值

```
1: #include <iostream>
2: int main()
3: {
4:     using std::cout;
5:     using std::endl;
6:
7:     int a=0, b=0, x=0, y=35;
```

```
8:  cout << "a: " << a << " b: " << b;
9:  cout << " x: " << x << " y: " << y << endl;
10: a = 9;
11: b = 7;
12: y = x = a+b;
13: cout << "a: " << a << " b: " << b;
14: cout << " x: " << x << " y: " << y << endl;
15: return 0;
16: }
```

▼ 输出:

```
a: 0 b: 0 x: 0 y: 35
a: 9 b: 7 x: 16 y: 16
```

▼ 分析:

在第 7 行，程序声明并初始化了 4 个变量；第 8 行和第 9 行打印了它们的值。在第 10 行，将 9 赋给了 a；在第 11 行将 7 赋给了 b。第 12 行将 a 和 b 的值相加，并将结果赋给 x。表达式 x=a+b 的结果为一个值（a 与 b 的和），这个值被赋给了 y。在第 13 行和第 14 行，打印了这 4 个变量的值，以验证上述结果。

### 5.3 使用运算符

运算符是一种让编译器执行某种操作的符号。运算符作用于操作数，在 C++ 中，任何表达式都可用作操作数。在 C++ 中，运算符分几类，读者将学习的前两类运算符是：

- 赋值运算符；
- 数学运算符。

#### 5.3.1 赋值运算符

赋值运算符 (=) 读者已经见过，它将左边的操作数的值修改为右边的表达式的值。下面的表达式将 a 加 b 的结果赋给操作数 x。

```
x = a + b;
```

左值和右值

可放在赋值运算符左边的操作数称为左值，可放在右边的称为右值。

注意，所有的左值都是右值，但并非所有的右值都是左值，例如，字面常量是右值，但不是左值。因此，可以编写下面的代码：

x = 5;

但不能编写下面的代码：

5 = x;

x 可以用作左值或右值，但 5 只能用作右值。

注意

常量是右值。由于它们的值不能修改，因此不能放在赋值运算符的左边，这意味着它们不是左值。

#### 5.3.2 数学运算符

另一类运算符是数学运算符，5 个数学运算符是加 (+)、减 (-)、乘 (\*)、除 (/) 和求模 (%)。

加法和减法运算符的工作原理与读者预期的相同：将两个数字相加或相减。乘法的工作原理相同，但乘法运算符是星号（\*）。除法运算符为斜杠。下面的示例表达式使用了所有这些运算符。在每个表达式中，结果都被赋给变量 `result`；右边的注释指出了 `result` 的值：

```
result = 56 + 32    // result = 88
result = 12 - 10    // result = 2
result = 21 / 7     // result = 3
result = 12 * 4     // result = 48
```

### 减法运算存在的问题

对 `unsigned` 整型变量执行减法运算时，如果结果为负，将出现奇怪的结果。这与第 3 章介绍的变量溢出时的情形类似。程序清单 5.2 说明了将较小的无符号数与较大的无符号数相减时出现的情况。

程序清单 5.2 减法和整型变量溢出

---

```
1: // Listing 5.2 - demonstrates subtraction and
2: // integer overflow
3: #include <iostream>
4:
5: int main()
6: {
7:     using std::cout;
8:     using std::endl;
9:
10:    unsigned int difference;
11:    unsigned int bigNumber = 100;
12:    unsigned int smallNumber = 50;
13:
14:    difference = bigNumber - smallNumber;
15:    cout << "Difference is: " << difference;
16:
17:    difference = smallNumber - bigNumber;
18:    cout << "\nNow difference is: " << difference << endl;
19:    return 0;
20: }
```

---

#### ▼ 输出：

```
Difference is: 50
Now difference is: 4294967246
```

#### ▼ 分析：

第 14 行使用了减法运算符，第 15 行打印了运算结果，这与预料的一致。第 17 行再次使用了运算符，但将一个较小的无符号数减去一个较大的无符号数。结果本应为负，但由于结果被赋给一个无符号变量，因此将发生第 3 章介绍的溢出。附录 C 对这一主题有详细介绍。

### 5.3.3 整数除法和求模

整数除法与小学学习的情况相同。将 21 除以 4 (21/4) 时，执行的就是整数除法，结果为 5 (余 1)。

第 5 个数学运算符对读者来说可能是新的。求模运算符 (%) 用于计算整数除法的余数。为计算 21 除以 4 的余数，可对 21 和 4 求模 (21%4)，结果为 1。

求模很有用。例如，你可能想每执行 10 次操作就打印一条语句。任何 10 的倍数对 10 取模后的结果均为 0。也就是说，1%10 为 1，2%10 为 2，依次类推，直至 10%10，其结果为 0。而 11%10 又回到 1，这种模式不断持续下去，直至下一个 10 的倍数 20，20%10 的结果再次为 0。在第 7 章讨论循环时将使用这种技术。



## FAQ

我将 5 除以 3 时，结果为 1，请问哪里出现了问题？

答：将两个整数相除时，结果仍为整数。

因此 5/3 的结果为 1（正确的答案是，结果为 1 余 2。要得到余数，可使用 5%3，其结果为 2。

要得到小数结果，必须使用浮点数（类型 float、double 或 long double）。

5.0/3.0 的结果为 1.66667。

如果除数或被除数为浮点数，编译器将生成浮点商。然而，将结果赋给一个整型左值时，结果将再次被截短。

## 5.4 结合使用赋值运算符与数学运算符

经常需要将一个值和一个变量相加，并将结果赋给该变量。如果有变量 myAge，要将其值加 2，可以编写这样的代码：

```
int myAge = 5;
int temp;
temp = myAge + 2;    // add 5 + 2 and put it in temp
myAge = temp;        // put it back in myAge
```

前两行创建变量 myAge 和一个临时变量。第三行将变量 myAge 的值和 2 相加，并将结果赋给 temp。在接下来的一行中，将这个值放回到 myAge 中，从而更新该变量。

然而，这种方法既迂回又浪费。在 C++ 中，可以在赋值运算符两边使用同一个变量，因而前面的代码变为：

```
myAge = myAge + 2;
```

这更好，也更清晰。在代数学中，该表达式毫无意义，但在 C++ 中，它的意思是：将 myAge 和 2 相加，并将结果赋给 myAge。

还有一种更简单但更难懂的写法：

```
myAge += 2;
```

这行代码使用了自赋值加运算符（+=）。这种运算符将右值与左值相加，并将结果重新赋给左值。如果 myAge 原来的值为 24，则执行上述语句后，其值为 26。

另外，还有自赋值减法运算符（-=）、自赋值乘法运算符（\*=）、自赋值除法运算符（/=）、自赋值求模运算符（%=）。

## 5.5 递增和递减

相加（相减）后，再将结果赋给变量时，最常用的值是 1。在 C++ 中，增加 1 被称为递增，减 1 被称为递减。C++ 有专门用于执行这些运算的运算符。

递增运算符（++）将变量的值加 1；递减运算符（--）将变量的值减 1。因此，如果有变量 Counter，要对其进行递增，可使用如下语句：

```
Counter++;           // Start with Counter and increment it.
```

该语句与如下更冗长的语句等价：

```
Counter = Counter + 1;
```

也与如下中等冗长的语句等价：

```
Counter += 1;
```

### 注意

正如读者可能已经猜到的，名称 C++ 意味着对其前身 C 执行递增运算，其含义是 C++ 在 C 语言的基础上进行了增量改进。

## 前缀和后缀

递增运算符 (++) 和递减运算符 (--) 有两种版本：前缀和后缀。前缀版本是将运算符放在变量名前 (++myAge)，后缀版本是放在变量名后 (myAge++)。

在简单语句中，使用前缀还是后缀版本无关紧要；但在对变量进行递增（递减），然后将结果赋给另一个变量这样的复杂语句中，它们的区别将相当大。使用前缀运算符时，在赋值前进行递增或递减；使用后缀运算符时，在赋值后进行递增或递减。前缀的语义是：将变量递增，然后再使用它；后缀的语义不同：先使用变量，然后再递增。

这一点刚开始可能让你感到糊涂，但通过下例你就明白了。如果整型变量 x 的值为 5，并使用前缀运算符编写了如下代码：

```
int a = ++x;
```

这告诉编译器，先将 x 递增（其值变为 6），再将该变量的值赋给 a。这样，执行该语句后，a 和 x 的值均为 6。

如果执行上述操作后，使用后缀运算符编写了如下语句：

```
int b = x++;
```

该语句告诉编译器，先将 x 的值（6）赋给 b，再将 x 递增。因此，执行该语句后，b 的值为 6，x 的值为 7。程序清单 5.3 说明了前缀和后缀运算的用法和含义。

程序清单 5.3 前缀和后缀运算符

```
1: // Listing 5.3 - demonstrates use of
2: // prefix and postfix increment and
3: // decrement operators
4: #include <iostream>
5: int main()
6: {
7:     using std::cout;
8:
9:     int myAge = 39;    // initialize two integers
10:    int yourAge = 39;
11:    cout << "I am: " << myAge << " years old.\n";
12:    cout << "You are: " << yourAge << " years old\n";
13:    myAge++;           // postfix increment
14:    ++yourAge;         // prefix increment
15:    cout << "One year passes...\n";
16:    cout << "I am: " << myAge << " years old.\n";
17:    cout << "You are: " << yourAge << " years old\n";
18:    cout << "Another year passes\n";
19:    cout << "I am: " << myAge++ << " years old.\n";
20:    cout << "You are: " << ++yourAge << " years old\n";
21:    cout << "Let's print it again.\n";
22:    cout << "I am: " << myAge << " years old.\n";
23:    cout << "You are: " << yourAge << " years old\n";
24:    return 0;
25: }
```

### ▼ 输出：

```
I am      39 years old
You are   39 years old
One year passes
I am      40 years old
You are   40 years old
Another year passes
I am      40 years old
You are   41 years old
Let's print it again
I am      41 years old
You are   41 years old
```

### ▼ 分析:

在第 9 行和第 10 行, 声明了两个整型变量, 并将它们都初始化为 39。第 11 行和第 12 行打印了这两个变量的值。

在第 13 行, 使用后缀递增运算符将 myAge 递增; 在第 14 行, 使用前缀递增运算符将 yourAge 递增。第 16 行和第 17 行打印结果, 它们的值相同, 都是 40。

在第 19 行, 在打印语句中, 使用后缀递增运算符将 myAge 递增。由于是后缀运算符, 递增发生在打印后, 因此再次打印 40, 然后将变量 myAge 递增。而在第 20 行, 使用前缀递增运算符将 yourAge 递增, 因此递增在打印前进行, 显示的值为 41。

最后, 在第 22 行和第 23 行, 再次打印两个变量的值。由于递增运算已经完成, 因此 myAge 的值为 41, 与 yourAge 的值相同。

## 5.6 理解运算符优先级

对于下述复杂语句:

```
x = 5 + 3 * 8;
```

先执行加法还是乘法运算呢? 如果先执行加法, 结果为  $8*8$  (64); 如果先执行乘法, 结果为  $5 + 24$  (29)。

C++标准对运算顺序做了规定。每个运算符都有优先级, 详情请参阅附录 C。乘法的优先级比加法高, 因此上述表达式的值为 29。

当两个数字运算符的优先级相同时, 执行顺序为自左至右。因此, 对于下面的语句:

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

先自左至右执行乘法。  $8*9=72$ ,  $6*4=24$ , 因此该表达式变成:

```
x = 5 + 3 + 72 + 24;
```

现在自左至右执行加法,  $5+3=8$ ,  $8+72=80$ ,  $80+24=104$ 。

有些运算符, 如赋值运算符 (=), 其执行顺序为自右至左。

如果优先级不能满足需求怎么办呢? 请看如下表达式:

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

在这个表达式中, 你不想将变量 NumMinutesToType 乘以 60, 再加上 NumMinutesToThink, 而是想先将这两个变量相加得到总分钟数, 然后再乘以 60 得到总秒数。

在这种情况下, 可以使用括号来改变优先级顺序。用括号括起的项的优先级比任何数学运算符都高。因此, 上例应改成这样:

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

## 5.7 括号的嵌套

对于复杂的表达式, 可能需要将一对括号嵌套到另一对括号内。例如, 可能要首先计算总秒数和参加人数, 再求其乘积:

```
TotalPersonSeconds = ( ( (NumMinutesToThink + NumMinutesToType) * 60) *  
(PeopleInTheOffice + PeopleOnVacation) )
```

这个复杂表达式将由内向外计算。首先, 将最内层括号中的 NumMinutesToThink 和 NumMinutesToType 相加, 并将结果乘以 60; 然后, 将 PeopleInTheOffice 与 PeopleOnVacation 相加, 并将结果与总秒数相乘。

这个例子提出了一个相关的问题。上述表达式虽然对于计算机来说很易理解, 但对于人来讲难以阅读、理解和修改。使用一些临时整型变量, 可将该表达式修改如下:

```
TotalMinutes = NumMinutesToThink + NumMinutesToType;
TotalSeconds = TotalMinutes * 60;
TotalPeople = PeopleInTheOffice + PeopleOnVacation;
TotalPersonSeconds = TotalPeople * TotalSeconds;
```

修改后，需要输入的代码更多，使用的临时变量也更多，但更容易理解得多。如果在代码前加上注释，对其功能进行解释，并用符号常量代替 60，代码将易于理解和维护。

应该	不应该
<div>请切记表达式都有值。</div> <div>要先对变量进行递增或递减，然后使用其值，请使用前缀运算符。</div> <div>要先使用变量的值，然后进行递增或递减，请使用后缀运算符。</div> <div>使用括号来改变优先顺序。</div>	<div>不要过深地嵌套括号，否则表达式将难以理解和维护。</div>

### 5.8 真值的本质

可以将每个表达式的值视为真或假。如果表达式为零，则返回 false，否则返回 true。  
在早期的 C++ 版本中，真和假都用整数表示，新的 ANSI 标准引入了 bool 类型。bool 变量只有两个可能的取值：false 或 true。

注意

以前很多编译器也提供了 bool 类型，它在内部表示为 long int，因此占用 4 字节。现在，遵循 ANSI 标准的编译器提供的 bool 类型通常只占 1 字节。

### 关系运算符

关系运算符用来对两个数字进行比较，判断它们相等、前者大于后者，还是前者小于后者。关系语句的值要么为 true，要么为 false。表 5.1 列出了关系运算符。

注意

所有关系运算符都返回一个 bool 值，即 true 或 false。在老版本的 C++ 中，这些运算符要么返回 0（表示 false），要么返回非零值（通常为 1，表示 true）。

如果整型变量 myAge 和 yourAge 的值分别为 45 和 50，可以用关系运算符==来判断它们是否相等：

```
myAge == yourAge; // is the value in myAge the same as in yourAge?
```

该表达式的值为 false，因为这两个变量不相等。同样，可以使用下面的表达式来判断 myAge 是否小于 yourAge：

```
myAge < yourAge; // is myAge less than yourAge?
```

该表达式的结果为 true，因为 45 小于 50。

警告

很多 C++ 初学者将赋值运算符(=)与相等运算符(==)混为一谈，这将在程序中造成重大错误。

6 个关系运算符分别是：等于(==)、小于(<)、大于(>)、小于等于(<=)、大于等于(>=)、不等于(!=)。表 5.1 列出了每个关系运算符及其使用示例。

表 5.1 关系运算符

名称	运算符	示例	结果
等于	==	100 == 50;	false
不等	!=	50 == 50;	true
大于	>	100 != 50;	true
大于等于	>=	50 != 50;	false
小于	<	100 > 50;	true
小于等于	<=	50 > 50;	false

应该	不应该
请切记，关系运算符返回 true 或 false。	不要将赋值运算符(=)与关系运算符等于(==)混为一谈。这是最常见的 C++编程错误之一，一定要当心。

5.9 if 语句

通常情况下，程序按语句在源代码中出现的顺序逐行执行。if 语句让你能够测试某个条件（如两个变量是否相等），然后根据结果，执行不同的代码片段。

最简单的 if 语句如下所示：

```
if (expression)
    statement;
```

括号中的 expression 可以是任何表达式，但通常都包含一个关系表达式。如果表达式的值为 false，则跳过下条语句。如果表达式的值为 true，则执行该语句。请看下面这个例子：

```
if (bigNumber > smallNumber)
    bigNumber = smallNumber;
```

上述代码对 bigNumber 和 smallNumber 进行比较。如果 bigNumber 更大，则执行第 2 行代码，将 smallNumber 的值赋给它；否则，跳过这行代码。

由于用大括号括起来的语句块相当于一句话，因此分支可能很大，且功能很强大：

```
if (expression)
{
    statement1;
    statement2;
    statement3;
}
```

下面是一个这种用法的简单例子：

```
if (bigNumber > smallNumber)
{
    bigNumber = smallNumber;
    std::cout << "bigNumber: " << bigNumber << "\n";
    std::cout << "smallNumber: " << smallNumber << "\n";
}
```

如果 bigNumber 比 smallNumber 大，不仅将 smallNumber 的值赋给 bigNumber，还将打印一条消息。程序清单 5.4 是一个更详细的基于关系运算符执行不同分支的例子。

程序清单 5.4 基于关系运算符进行分支

```
1: // Listing 5.4 - demonstrates if statement
2: //,used with relational operators
3: #include <iostream>
```



```
4: int main()
5: {
6:     using std::cout;
7:     using std::cin;
8:
9:     int MetsScore, YankeesScore;
10:    cout << "Enter the score for the Mets: ";
11:    cin >> MetsScore;
12:
13:    cout << "\nEnter the score for the Yankees: ";
14:    cin >> YankeesScore;
15:
16:    cout << "\n";
17:
18:    if (MetsScore > YankeesScore)
19:        cout << "Let's Go Mets!\n";
20:
21:    if (MetsScore < YankeesScore)
22:    {
23:        cout << "Go Yankees!\n";
24:    }
25:
26:    if (MetsScore == YankeesScore)
27:    {
28:        cout << "A tie? Naah, can't be.\n";
29:        cout << "Give me the real score for the Yanks: ";
30:        cin >> YankeesScore;
31:
32:        if (MetsScore > YankeesScore)
33:            cout << "Knew it! Let's Go Mets!";
34:
35:        if (YankeesScore > MetsScore)
36:            cout << "Knew it! Go Yanks!";
37:
38:        if (YankeesScore == MetsScore)
39:            cout << "Wow, it really was a tie!";
40:    }
41:
42:    cout << "\nThanks for telling me.\n";
43:    return 0;
44: }
```

#### ▼ 输出:

Enter the score for the Mets: 10

Enter the score for the Yankees: 10

A tie? Naah, can't be

Give me the real score for the Yanks: 8

Knew it! Let's Go Mets!

Thanks for telling me.

#### ▼ 分析:

该程序要求用户输入两个棒球队的得分，它们被保存在整型变量 `MetsScore` 和 `YankeesScore`。在第 18、21 和 26 行，使用 `if` 语句对这两个变量进行比较。

如果一个得分比另一个高，则打印一条消息。如果得分相等，则执行第 27~40 行的代码块：再次要求用户输入得分，并对它们进行比较。

注意，如果一开始 `Yankee` 队的得分就比 `Mets` 队高，第 18 行的 `if` 语句的值将为 `false`，因此第 19 行不会被执行。在这种情况下，第 21 行的测试将为 `true`，并执行第 23 行的语句。接下来，测试第 26 行的 `if` 语句，而其值为 `false`，因此程序跳过整个语句块，执行第 41 行。这个示例表明，`if` 语句的值为 `true` 并不会导致程序跳过后续的 `if` 语句。

前两条 `if` 语句的对应的操作代码都只有 1 行（打印 `Let's Go Mets!` 或 `Go Yankees!`）。在第一条 `if` 语句（18 行）中，对应的操作代码没有用大括号括起，因为单行代码块不需要。然而，用大括号括起也

是合法的，如第 22~24 行所示。

#### 避免使用 if 语句时的常见错误

许多 C++ 新手会无意中在 if 语句后面加一个分号：

```
if(SomeValue < 10);    // Oops! Notice the semicolon!
    SomeValue = 10;
```

程序员的本意是测试 SomeValue 是否小于 10，如果是，将其值设置为 10，从而使 SomeValue 的最小值为 10。如果运行上述语句片段，将发现 SomeValue 的值总为 10。为什么会这样呢？原因就在于在 if 语句后面加了一个分号（什么也不做的运算符）。

还记得吗，缩进对编译器来说没有任何意义。上面这段代码可以更准确地写成下面这样：

```
if (SomeValue < 10) // test
; // do nothing
SomeValue = 10; // assign
```

删除分号将使最后一行成为 if 语句的一部分，这样代码按程序员的本意执行。

为最大限度地减少出现这种问题的机会，可以在编写 if 语句时总是使用大括号，即使 if 语句体只有 1 行：

```
if (SomeValue < 10)
{
    SomeValue = 10;
};
```

### 5.9.1 缩进风格

程序清单 5.4 演示了一种 if 语句缩进风格。然而，如果询问一组程序员，最佳的大括号对齐风格是什么，无疑会爆发一场宗教战争。尽管目前有十几种风格，但最流行的是以下 3 种：

- 将左大括号放在条件后，右大括号与 if 对齐：

```
if (expression){
    statements
}
```

- 将左、右大括号均与 if 对齐，并缩进语句：

```
if (expression)
{
    statements
}
```

- 缩进大括号和语句：

```
if (expression)
{
    statements
}
```

本书采用第二种风格，因为将大括号和测试条件对齐时，更容易知道语句块从何处开始到何处结束。同样，选用哪种风格关系并不大，只要保持一致即可。

### 5.9.2 else 语句

程序经常需要在条件为真时执行一个分支，而条件为假时执行另一个分支。在程序清单 5.4 中，如果第一个条件 (MetsScore > Yankees score) 为真，将打印一条消息 (Let's Go Mets!)，如果该条件为假，将打印另一条消息 (Go Yankees!)。

迄今为止使用的方法（先测试一个条件然后再测试另一个条件）是可行的，但有点啰嗦。使用关键字 else 可使代码的可读性更好：

```
if (expression)
    statement;
else
    statement;
```

程序清单 5.5 演示了关键字 `else` 的用法。

#### 程序清单 5.5 关键字 `else` 的用法

```
1: // Listing 5.5 - demonstrates if statement
2: // with else clause
3: #include <iostream>
4: int main()
5: {
6:     using std::cout;
7:     using std::cin;
8:
9:     int firstNumber, secondNumber;
10:    cout << "Please enter a big number: ";
11:    cin >> firstNumber;
12:    cout << "\nPlease enter a smaller number: ";
13:    cin >> secondNumber;
14:    if (firstNumber > secondNumber)
15:        cout << "\nThanks!\n";
16:    else
17:        cout << "\nOops. The first number is not bigger!";
18:
19:    return 0;
20: }
```

#### ▼ 输出:

Please enter a big number: 10

Please enter a smaller number: 12

Oops. The first number is not bigger!

#### ▼ 分析:

执行第 14 行的 `if` 语句。如果条件为真，执行第 15 行的语句，然后程序流程进入第 18 行 (`else` 语句的后面)；如果第 14 行的条件为假，则进入 `else` 子句，执行第 17 行的语句。如果删掉第 16 行的 `else` 子句，则无论 `if` 语句的条件是真还是假，程序都将执行第 17 行的语句。

请记住，`if` 语句在第 15 行结束。如果没有 `else` 子句，第 17 行将是程序的下一条语句。另外，`if` 语句体和 `else` 语句体都可以是用大括号括起的代码块。

#### If 语句

`if` 语句的语法有如下两种形式:

第 1 种形式:

```
if (expression)
    statement;
next_statement;
```

如果 `expression` 为真，将执行 `statement`，再执行 `next_statement`；如果为假，将不执行 `statement`，而直接执行 `next_statement`。

请记住，`statement` 可以是以分号结尾的单条语句，也可以是用大括号括起的代码块。

第 2 种形式:

```
if (expression)
    statement1;
else
    statement2;
next_statement;
```

如果 `expression` 为真，则执行 `statement1`；否则执行 `statement2`。然后，程序继续执行 `next_statement`。

示例 1:

```
Example
if (SomeValue < 10)
    cout << "SomeValue is less than 10";
else
    cout << "SomeValue is not less than 10!";
cout << "Done." << endl;
```

### 5.9.3 高级 if 语句

值得注意的是，任何语句都可用于 if 或 else 子句中，甚至可以是另一条 if 或 else 语句。因此你可能看到下面这种形式的复杂 if 语句：

```
if (expression1)
{
    if (expression2)
        statement1;
    else
    {
        if (expression3)
            statement2;
        else
            statement3;
    }
}
else
    statement4;
```

这个复杂 if 语句的意思是：如果 expression1 和 expression2 均为真，则执行 statement1。如果 expression1 为真而 expression2 为假，且 expression3 为真，则执行 statement2。如果 expression1 为真，而 expression2 和 expression3 均为假，则执行 statement3。最后，如果 expression1 为假，则执行 statement4。正如读者看到的，复杂 if 语句很容易令人迷惑。

程序清单 5.6 给出了一个复杂 if 语句的例子。

程序清单 5.6 嵌套的复杂 if 语句

```
1: // Listing 5.6 - a complex nested
2: // if statement
3: #include <iostream>
4: int main()
5: {
6:     // Ask for two numbers
7:     // Assign the numbers to bigNumber and littleNumber
8:     // If bigNumber is bigger than littleNumber,
9:     // see if they are evenly divisible
10:    // If they are, see if they are the same number
11:
12:    using namespace std;
13:
14:    int firstNumber, secondNumber;
15:    cout << "Enter two numbers.\nFirst: ";
16:    cin >> firstNumber;
17:    cout << "\nSecond: ";
18:    cin >> secondNumber;
19:    cout << "\n\n";
20:
21:    if (firstNumber >= secondNumber)
22:    {
23:        if ( (firstNumber % secondNumber) == 0) // evenly divisible?
24:        {
25:            if (firstNumber == secondNumber)
26:                cout << "They are the same!\n";
27:            else
28:                cout << "They are evenly divisible!\n";
29:        }
```

```

30:         else
31:             cout << "They are not evenly divisible!\n";
32:     }
33:     else
34:         cout << "Hey! The second one is larger!\n";
35:     return 0;
36: }

```

**▼ 输出:**

```

Enter two numbers.
First: 10

Second: 2
They are evenly divisible!

```

**▼ 分析:**

该程序提示用户输入两个数，每次输入一个，然后对它们进行比较。第 21 行的第 1 条 if 语句检查第一个数是否大于等于第 2 个数。如果不是，则执行第 33 行的 else 子句。

如果第 1 条 if 语句为真，则执行从第 22 行开始的代码块，并测试第 23 行的第 2 条 if 语句。这条 if 语句检查将第 1 个数除以第 2 个数时，余数是否为 0。如果是，这两个数要么相等，要么前者是后者的整数倍。第 25 行的第 3 条 if 语句检查两数是否相等，并根据结果打印相应的消息。

如果第 23 行的 if 语句为假，将执行第 30 行的 else 子句。

## 5.10 在嵌套 if 语句中使用大括号

虽然 if 语句体只有一条语句时，不使用大括号也完全合法，对嵌套 if 语句也如此，但这样做将令人迷惑。下面的代码在 C++ 中是完全合法的，但看起来让人感到迷惑：

```

if (x > y)           // if x is bigger than y
    if (x < z)        // and if x is smaller than z
        x = y;       // set x to the value in y
    else             // otherwise, if x isn't less than z
        x = z;       // set x to the value in z
else                 // otherwise if x isn't greater than y
    y = x;           // set y to the value in x

```

空格和缩进只是为程序员提供方便，对编译器来说毫无意义。如果不使用大括号，很容易对其中的逻辑感到迷惑，将 else 与 if 的对应关系搞错。程序清单 5.7 演示了这种问题。

**程序清单 5.7 演示为什么大括号有助于阐明 else 语句与 if 语句的对应关系**

```

1: // Listing 5.7 - demonstrates why braces
2: // are important in nested if statements
3: #include <iostream>
4: int main()
5: {
6:     int x;
7:     std::cout << "Enter a number less than 10 or greater than 100: ";
8:     std::cin >> x;
9:     std::cout << "\n";
10:
11:     if (x >= 10)
12:         if (x > 100)
13:             std::cout << "More than 100, Thanks!\n";
14:         else // not the else intended!
15:             std::cout << "Less than 10, Thanks!\n";
16:
17:     return 0;
18: }

```

**▼ 输出:**

```

Enter a number less than 10 or greater than 100: 20

Less than 10, Thanks!

```



**▼ 分析:**

程序员的本意是要用户输入一个小于 10 或大于 100 的数, 然后检查输入的值是否正确, 并打印一条致谢消息。

如果第 11 行的 if 语句为真, 将执行第 12 行的语句。在这个例子中, 当输入的数字大于 10 时, 将执行第 12 行的语句。第 12 行也包含一条 if 语句; 如果输入的数大于 100, 该 if 语句为真。如果输入的数大于 100, 将执行第 13 行的语句: 打印一条相应的消息。

如果输入的数小于 10, 第 11 行的 if 语句将为假。程序将跳到该 if 语句后面的语句 (这里为第 16 行) 处执行。如果用户输入一个比 10 小的数, 将出现如下输出:

```
Enter a number less than 10 or greater than 100: 9
```

正如读者看到的, 没有打印消息。程序员的本意是, 第 14 行的 else 与第 11 行的 if 匹配, 并进行了相应的缩进。不幸的是, 编译器将这个 else 与第 12 行的 if 相对应, 于是出现了上述微妙的错误。

之所以称为微妙的错误, 是因为编译器没有指出。这是一个合法的 C++ 程序, 只不过没有实现程序员的意图。另外, 程序员测试该程序时, 往往看似能正常运行: 只要输入的数大于 10, 程序将看似一切正常。然而, 如果输入一个 11~99 的数, 将发现程序明显存在问题! 程序清单 5.8 通过加入必要的大括号, 解决了这个问题。

**程序清单 5.8 在 if 语句中正确地使用大括号**

```
1: // Listing 5.8 - demonstrates proper use of braces
2: // in nested if statements
3: #include <iostream>
4: int main()
5: {
6:     int x;
7:     std::cout << "Enter a number less than 10 or greater than 100: ";
8:     std::cin >> x;
9:     std::cout << "\n";
10:
11:     if (x >= 10)
12:     {
13:         if (x > 100)
14:             std::cout << "More than 100, Thanks!\n";
15:     }
16:     else // fixed!
17:         std::cout << "Less than 10, Thanks!\n";
18:     return 0;
19: }
```

**▼ 输出:**

```
Enter a number less than 10 or greater than 100: 9
Less than 10, Thanks!
```

**▼ 分析:**

第 12 行和第 15 行的一对大括号使得它们之间的所有代码成为一条语句。现在, 第 16 行的 else 与第 11 行的 if 相匹配, 这与程序员的本意相符。

如果用户输入 9, 第 11 行的 if 语句将为真; 然而第 13 行的 if 语句为假, 因此不打印任何消息。如果在第 14 行后再加一个 else 子句, 以捕获输入错误并打印一条消息, 程序将更完美。

**提示**

可以在 if 和 else 子句中使用大括号 (即使条件后面只有一条语句), 以最大限度地减少 if...else 语句可能带来的问题:

```
if (SomeValue < 10)
{
    SomeValue = 10;
}
else
{
    SomeValue = 25;
};
```

注意

本书所有的程序都是针对要讨论的问题而编写的。因此都很简单，没有包含防范用户操作错误的代码。在专业品质的代码中，应考虑到用户可能犯的所有操作错误，并妥善进行处理。

5.11 使用逻辑运算符

经常需要同时问多个关系型问题，例如，x 大于 y 且 y 大于 z 吗？程序可能需要确定这两个条件（或一组其他的条件）都为真时，才执行某项操作。

假设一个复杂的警报系统，其逻辑为：如果在非节假日的下午 6 点以后或周末，大门警报器响起，则报警。C++ 提供了 3 个逻辑运算符，用于完成这种判断，如表 5.2 所示。

表 5.2 逻辑运算符

运算符	符号	示例
AND	&&	<i>expression1 &amp;&amp; expression2</i>
OR		<i>expression1    expression2</i>
NOT	!	<i>!expression</i>

5.11.1 逻辑 AND 运算符

逻辑 AND 语句使用 AND 运算符来连接和计算两个表达式的值，如果两个表达式均为真，逻辑 AND 语句也为真。如果你饿了，且身上有钱，则会买午餐。因此，如果 x 和 y 都为 5，在下面的语句为真；如果任何一个不为 5，则为假：

```
if ( (x == 5) && (y == 5) )
```

注意，要使整个表达式为真，&& 两边都必须为真。

逻辑 AND 运算符为 &&。单个 & 是另一种运算符——按位 AND。

5.11.2 逻辑 OR 运算符

逻辑 OR 语句也对两个表达式进行计算。如果任何一个为真，则整个表达式为真。如果有现钱或信用卡，可以付账。不必既有现金又有信用卡，而只需有一种就行了，尽管两者都有也行。因此如果 x 或 y 等于 5，或者 x、y 均等于 5，下面 if 语句为真：

```
if ( (x == 5) || (y == 5) )
```

逻辑 OR 运算符为 ||。单个 | 是另一种运算符——按位 OR。

5.11.3 逻辑 NOT 运算符

如果被测试的表达式为假，则逻辑 NOT 语句为真。同样，如果表达式为假，则逻辑 NOT 测试为真。因此，仅当 x 不等于 5 时，下面的测试才为真

```
if ( !(x == 5) )
```

该语句与下面的语句等价：

```
if ( x != 5 )
```

5.12 简化求值

编译器计算如下 AND 语句的值时，首先判断第一个表达式(x == 5)是否为真，如果为假（即 x 不

等于 5), 则不再判断第二个表达式( $y=5$ )是否为真, 因为仅当两个表达式都为真时, AND 语句才为真。

```
if ( (x == 5) && (y == 5) )
```

同样, 如果编译器计算如下所示的 OR 语句的值时, 如果第一个表达式( $x=5$ )为真, 编译器将不再计算第二个表达式的值, 因为任何一个表达式为真, 整个 OR 语句就为真。

```
if ( (x == 5) || (y == 5) )
```

这虽然看似不重要, 但请看下面的示例:

```
if ( (x == 5) || (++y == 3) )
```

如果  $x$  不为 5, 将不计算表达式( $++y == 3$ )。如果程序员指望无论在什么情况下都将  $y$  递增, 这种愿望将落空。

## 5.13 关系运算符的优先级

和所有 C++ 表达式一样, 使用关系运算符和逻辑运算符时, 都将返回一个值: true 或 false。和所有表达式一样, 它们都有优先级顺序 (参见附录 C), 这决定了先计算哪个关系的结果。在计算下述语句的值时, 这一点非常重要:

```
if ( x > 5 && y > 5 || z > 5 )
```

程序员可能希望这样: 如果  $x$  和  $y$  均大于 5 或  $z$  大于 5, 则该表达式为真。另一方面, 程序员可能希望这样: 仅当  $x$  大于 5 且  $y$  或  $z$  大于 5 时, 该表达式的值为真。

如果  $x$  为 3,  $y$  和  $z$  均为 10, 且采用第一种解释, 则整个表达式为真 ( $z$  大于 5, 因此忽略  $x$  和  $y$ ); 而采用第二种解释时, 整个表达式为假 ( $x$  大于 5 为假, 因此  $\&\&$  右边是什么无关紧要, 因为两边都必须为真, 整个表达式才为真)。

虽然运算符的优先级决定了先计算哪个关系, 但使用括号可以改变优先级, 使语句更清晰:

```
if ( (x > 5) && (y > 5 || z > 5) )
```

使用前面的取值时, 表达式为假。由于  $x$  大于 5 为假, 即 AND 语句的左边为假, 因此整条语句为假。别忘了, 仅当 AND 运算符两边均为真时整个语句才为真: 如果某种东西的味道不好, 那么它不可能既好吃又对身体有益。

### 提示

通常使用括号, 可以阐明你要将什么组合起来, 这样做通常是个不错的主意。别忘了, 目标是编写能够正确运行且易于阅读和理解的程序。使用括号有助于阐明意图, 避免因错误地理解运算符的优先级带来的错误。

## 5.14 再谈真和假

在 C++ 中, 0 被解释为代表假, 其他值都被解释为真。表达式总是有一个值, 许多程序员在 if 语句中利用这一特点。例如, 下述语句的含义为, 如果  $x$  不为零, 将其设置为 0:

```
if (x)           // if x is true (nonzero)
    x = 0;
```

这有点玩技巧, 如果写成下面这样将更清楚:

```
if (x != 0)       // if x is not zero
    x = 0;
```

这两种写法都合法, 但后者更清楚。良好的编程习惯是, 用前一种方法来判断逻辑真和假, 而不使用它来判断非 0 值。

下面这两条语句也是等效的:

```
if (!x)           // if x is false (zero)
if (x == 0)       // if x is zero
```

但第 2 条语句更容易理解, 同时如果要判断的是  $x$  的数学值而不是逻辑状态, 这条语句更明确。

## 应该

请在逻辑测试中使用括号，使它们更清楚，优先顺序更明确。

务必在嵌套 if 语句中使用大括号，使 else 语句的对应关系更清楚并防止出错。

## 不应该

不要用 `if(x)` 来代替 `if(x!=0)`；因为后者更清晰。  
不要用 `if(!x)` 来代替 `if(x==0)`；因为后者更清晰。

## 5.15 条件运算符（三目运算符）

条件运算符（?:）是 C++ 中唯一一个三目运算符，即它是唯一一个需要 3 个操作数的运算符。

条件运算符接受 3 个表达式并返回一个值：

`(expression1) ? (expression2) : (expression3)`

其含义是：如果 `expression1` 为真，则返回 `expression2` 的值；否则返回 `expression3` 的值。通常，返回的值将赋给一个变量。程序清单 5.9 演示了如何使用条件运算符来代替 if 语句。

程序清单 5.9 条件运算符

```
1: // Listing 5.9 - demonstrates the conditional operator
2: //
3: #include <iostream>
4: int main()
5: {
6:     using namespace std;
7:
8:     int x, y, z;
9:     cout << "Enter two numbers.\n";
10:    cout << "First: ";
11:    cin >> x;
12:    cout << "\nSecond: ";
13:    cin >> y;
14:    cout << "\n";
15:
16:    if (x > y)
17:        z = x;
18:    else
19:        z = y;
20:
21:    cout << "After if test, z: " << z;
22:    cout << "\n";
23:
24:    z = (x > y) ? x : y;
25:
26:    cout << "After conditional test, z: " << z;
27:    cout << "\n";
28:    return 0;
29: }
```

### ▼ 输出：

Enter two numbers.  
First: 5

Second: 8

After if test, z: 8  
After conditional test, z: 8

### ▼ 分析：

创建了 3 个整型变量：x、y 和 z。前两个变量的值由用户提供。第 16 行的 if 语句检查 x 和 y 哪个大，然后将较大的值赋给 z。第 21 行打印这个值。

第 24 行的条件运算符进行同样的检查，并将较大的值赋给  $z$ 。这行代码的含义是：如果  $x$  大于  $y$ ，返回  $x$  的值；否则返回  $y$  的值；然后将返回的值赋给  $z$ 。第 26 行打印这个值。正如读者看到的，可用条件语句替代 `if...else` 语句，但它更简短。

## 5.16 总结

在本章中，读者学习了什么是 C++ 语句和表达式、C++ 运算符的作用以及 C++ `if` 语句的工作原理。在可以使用单条语句的任何地方，也可以使用一对大括号括起的语句块。

每个表达式都有一个值，可以使用 `if` 语句或条件运算符对其进行测试。另外，读者还学会了如何使用逻辑运算符计算多条语句的值，如何使用关系运算符对两个值进行比较以及如何使用赋值运算符进行赋值。

本章还介绍了运算符优先级。读者知道了如何使用括号来改变优先顺序，并使优先顺序更清晰，更容易理解。

## 5.17 问与答

问：在优先级能够确定运算符的执行顺序时，为什么还要使用不必要的括号？

答：虽然不用括号，编译器也知道优先级，而程序员也可以查阅优先级顺序，但通过使用括号，可使程序更容易理解，进而更容易维护。

问：如果关系运算符总是返回真或假，为什么要规定任何非零值都表示真？

答：这种约定是从 C 语言那里继承而来的，编写低级软件（如操作系统和实时控制软件）时经常会使用这种约定。这种用法可能成为检测掩码或变量的所有位是否为 0 的简捷方式。

虽然关系运算符返回真或假，但任何表达式都会返回一个值，在 `if` 语句中也可以对这些值进行计算。下面是一个这样的例子：

```
if ( (x = a + b) == 35 )
```

这是一条完全合法的 C++ 语句。即使  $a$  与  $b$  的和不等于 35，该表达式也会返回一个值。另外，无论什么情况下， $a$  与  $b$  的和都将被赋给  $x$ 。

问：制表符、空格、换行符对程序有何影响？

答：制表符、空格和换行符（被称为空白符）对程序没有影响，虽然明智地使用空白可以提高程序的可读性。

问：负数是真还是假？

答：所有非零值（无论正负）均为真。

## 5.18 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 5.18.1 测验

1. 什么是表达式？



2.  $x=5+7$  是表达式吗？它的值是多少？
3.  $201/4$  的值是多少？
4.  $201\%4$  的值是多少？
5. 如果 `myAge`、`a` 和 `b` 都是 `int` 变量，则执行下列代码后，它们的值是多少？

```
myAge = 39;  
a = myAge++;  
b = ++myAge;
```

6.  $8+2*3$  的值是多少？
7. `if(x = 3)`和 `if(x == 3)`有何不同？
8. 下列值为 `true` 还是 `false`？
  - a. 0
  - b. 1
  - c. -1
  - d. `x = 0`
  - e. `x == 0` //assume that `x` has the value of 0

### 5.18.2 练习

1. 编写一条 `if` 语句，检查两个整型变量，并将较大的变成较小的，只能使用一条 `else` 子句。
2. 查看下列程序。假定输入的 3 个数，并写出预期的输出。

```
1:  #include <iostream>  
2:  using namespace std;  
3:  int main()  
4:  {  
5:      int a, b, c;  
6:      cout << "Please enter three numbers\n";  
7:      cout << "a: ";  
8:      cin >> a;  
9:      cout << "\nb: ";  
10:     cin >> b;  
11:     cout << "\nc: ";  
12:     cin >> c;  
13:  
14:     if (c = (a-b))  
15:         cout << "a: " << a << " minus b: " << b <<  
16:             " equals c: " << c;  
17:     else  
18:         cout << "a-b does not equal c: ";  
19:     return 0;  
20: }
```

3. 输入练习 2 中的程序，然后编译、链接并运行它。输入 20、10 和 50。得到了预期的输出吗？为什么没有？

4. 查看下面的程序并预测输出。

```
1:  #include <iostream>  
2:  using namespace std;  
3:  int main()  
4:  {  
5:      int a = 2, b = 2, c;  
6:      if (c = (a-b))  
7:          cout << "The value of c is: " << c;  
8:      return 0;  
9:  }
```

5. 输入、编译、链接并运行练习 4 中的程序。输出是什么？为什么？

# 第 6 章

## 使用函数组织代码

虽然面向对象编程已经将注意力从函数转向对象，但函数仍然是任何程序的核心。全局函数位于对象和类之外，成员函数（也称成员方法）位于类内。

在本章中，您将学习：

- 函数及其组成部分
- 如何声明和定义函数
- 如何向函数传递参数
- 如何从函数返回一个值

本章介绍全局函数，第 10 章介绍类和对象内部的函数。

### 6.1 什么是函数

函数实际上是能够对数据进行处理并返回一个值的子程序。每个 C++ 程序都至少有一个函数：main()。程序启动时，main() 函数被自动调用。main() 函数可能调用其他函数，而有些被调用的函数又调用了其他函数。

由于这些函数不是对象的一部分，它们被称为全局的，即可在程序的任何地方访问它们。在本章中，除非特别声明，指的都是全局函数。

每个函数都有自己的名称，程序遇到函数名时，将执行函数体。这被称为调用函数。函数执行完（遇到 return 语句或函数末尾的大括号）后，程序回到函数调用的下一行继续执行。图 6.1 说明了这种流程。

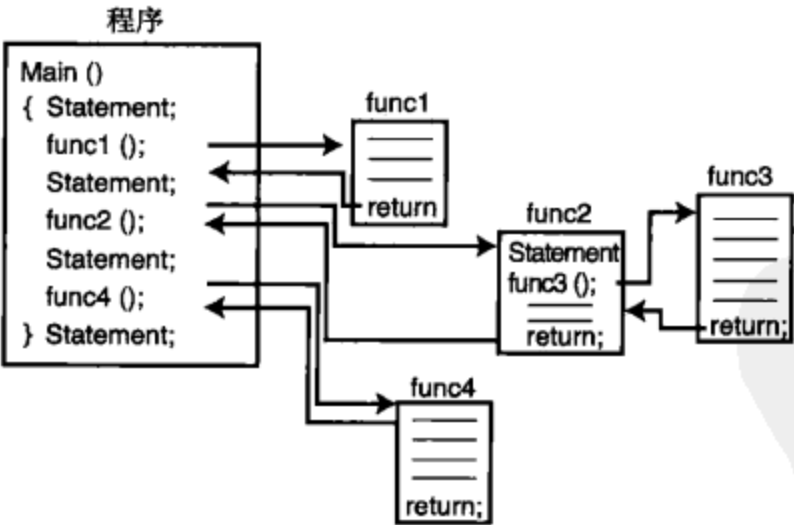


图 6.1 程序调用函数时将执行函数，然后返回到函数调用的下一行继续执行

设计良好的函数执行单个具体的、易于理解的任务，函数名指出了这种任务。复杂的任务应分成多个函数来完成，然后依次调用这些函数。

函数通常有两种类型：用户定义的函数和内置函数。内置函数是编译器软件包的一部分：由开发商提供给用户使用。用户定义的函数是用户自己编写的函数。

## 6.2 返回值、参数和实参

正如第2章指出的，函数可以接受值，还能够返回一个值。

调用函数时，它可以完成工作，并返回一个值作为工作的结果。这个值被称为返回值，返回值的类型必须声明。下面的代码声明了一个名为 `myFunction()` 的函数，它返回一个 `int` 值：

```
int myFunction();
```

接下来看下面的声明：

```
int myFunction(int someValue, float someFloat);
```

该声明指出，`myFunction` 也返回一个 `int` 值，但它还接受两个值。

将值传递给函数时，这些值将作为变量，可以在函数中对其进行操纵。对传入值的描述称为参数列表。在前一个例子中，参数列表中包含 `someValue` 和 `someFloat`，它们分别是 `int` 类型和 `float` 类型的变量。

正如读者看到的，参数描述了函数被调用时传递给它的值的类型。传递给函数的实际值被称为实参。请看下面的语句：

```
int theValueReturned = myFunction(5,6.7);
```

上述代码将 `int` 变量 `theValueReturned` 初始化为函数 `myFunction` 的返回值，同时 5 和 6.7 被作为实参传递给该函数。实参的类型必须与声明的参数类型匹配。在这个例子中，将 5 和 6.7 分别传递给一个 `int` 变量和一个 `float` 变量，因此类型是匹配的。

## 6.3 声明和定义函数

要在程序中使用函数，必须先声明函数然后再定义它。声明将函数的名称、返回值类型和参数告诉编译器；定义将函数的工作原理告诉编译器。

如果不声明，任何函数都不能被其他函数调用。函数的声明又被称为原型。

有 3 种声明函数的方法：

- 将函数原型放在文件中，然后使用编译指令 `#include` 将该文件包含到程序中，
- 将函数原型放在使用它的文件中，
- 在函数被其他函数调用前定义它。这样做时，定义将作为原型。

虽然可以在使用函数前定义它，从而避免创建函数原型，但这并不是好的编程习惯，原因有 3 个。

首先，要求函数在文件中以特定的顺序出现是不明智的。当需求发生变化时，这将使程序难以维护。

其次，在某些情况下，函数 `A()` 可能要调用函数 `B()`，而函数 `B()` 也要调用函数 `A()`。但不可能既在函数 `A()` 之前定义函数 `B()`，又在函数 `B()` 之前定义函数 `A()`。在这种情况下，至少需要声明其中一个函数。

最后，函数原型是一种优秀而强大的调试技术。如果原型指出函数将接受一组特定的参数或返回一个特定类型的值，当函数与原型不匹配时，编译器将指出错误，而不用等到运行程序时才显现出来。这像复式簿记。原型和定义相互检查，以减少输入错误导致程序错误出现问题的可能性。

虽然如此，但很多程序员还是选择第三种方式。这是因为这样可以减少代码行、简化维护工作（如果修改函数头，则必须修改原型），同时函数在文件中的顺序通常非常稳定。然而，在有些情况下，原型是必不可少的。

### 6.3.1 函数原型

您使用的很多内置函数的原型已经编写好，它们位于在程序中使用 `#include` 包含进来的文件中。对

于自己编写的函数，必须包含其原型。

函数原型是一条语句，这意味着它以分号结尾。它由函数的返回值类型和特征标组成。函数特征标包括函数名和参数列表。

参数列表是所有参数及其类型的列表，参数之间用逗号分开。图 6.2 说明了函数原型的组成部分。

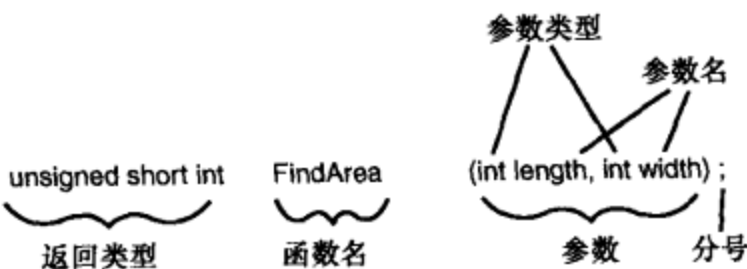


图 6.2 函数原型的组成部分

函数原型及其定义在返回类型和特征标方面必须完全相同；否则，将出现编译阶段错误。然而，函数原型可以不包含参数名，而只需包含参数类型。像下面这样的原型完全合法：

```
long Area(int, int);
```

该原型声明了一个名为 `Area()` 的函数，其返回类型为 `long` 且接受两个 `int` 参数。虽然这样做合法，但并非好主意。加入参数名可使原型更清晰。加入参数名后，该函数原型如下：

```
long Area(int length, int width);
```

现在，函数的功能和参数明显得多。

注意，所有函数都有一个返回值，如果没有明确指定，默认为 `int`。然而，明确地声明每个函数（包括 `main()`）的返回类型，可使程序更清晰。

如果函数不返回任何值，将其返回类型声明为 `void`，如下所示：

```
void printNumber( int myNumber);
```

这声明了一个名为 `printNumber` 的函数，它接受一个 `int` 参数。由于返回类型为 `void`，因此不返回任何值。

6.3.2 定义函数

函数定义由函数头和函数体组成。函数头类似于函数原型，只是必须包含参数名，且不以分号结尾。

函数体是用一对大括号括起的一组语句。图 6.3 说明了函数头和函数体。

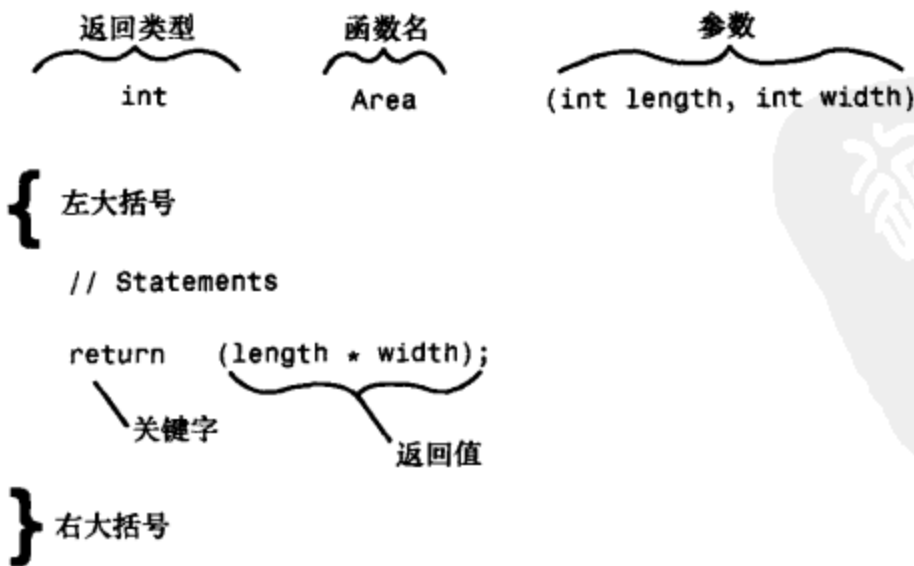


图 6.3 函数头和函数体

程序清单 6.1 中的程序包含了函数 `Area()` 的原型。

程序清单 6.1 函数的声明、定义和用法

```
1: // Listing 6.1 - demonstrates the use of function prototypes
2:
3: #include <iostream>
4: int Area(int length, int width); //function prototype
5:
6: int main()
7: {
8:     using std::cout;
9:     using std::cin;
10:
11:     int lengthOfYard = 0;
12:     int widthOfYard = 0;
13:     int areaOfYard = 0;
14:
15:     cout << "\nHow wide is your yard? ";
16:     cin >> widthOfYard;
17:     cout << "\nHow long is your yard? ";
18:     cin >> lengthOfYard;
19:
20:     areaOfYard= Area(lengthOfYard, widthOfYard);
21:
22:     cout << "\nYour yard is ";
23:     cout << areaOfYard;
24:     cout << " square feet\n\n";
25:     return 0;
26: }
27:
28: int Area(int len, int wid)
29: {
30:     return len * wid;
31: }
```

#### ▼ 输出:

```
How wide is your yard? 100
How long is your yard? 200
Your yard is 20000 square feet
```

#### ▼ 分析:

函数 `Area()` 的原型位于第 4 行。将其与第 28 行的函数定义比较将发现, 函数名、返回类型和参数类型都完全一致。如果它们不同, 将出现编译错误。事实上, 唯一的不同在于, 函数原型以分号结尾且没有函数体。

另外, 原型中的参数名为 `length` 和 `width`, 而定义中的参数名为 `len` 和 `wid`。正如以前讨论的, 原型中的参数名没有实际意义, 只是为程序员提供一些信息。良好的编程习惯是, 在原型和定义中使用相同的参数名, 但正如程序清单 6.1 表明的, 并非必须这样做。

实参按声明和定义中的参数排列顺序传递, 但不进行名称匹配。如果传递 `widthOfYard` 和 `lengthOfYard`, 函数 `Find Area()` 将把 `widthOfYard` 的值用作长度, 把 `lengthOfYard` 的值用作宽度。

#### 注意

函数体总是由大括号括起来, 即使像这个例子中那样只有一条语句。

## 6.4 函数的执行

调用函数时, 将从左大括号后的第一条语句开始执行。使用 `if` 语句可以提供分支 (`if` 及其他相关语句将在第 7 章介绍)。函数还可以调用其他函数甚至自己 (参见 6.12.2 节)。

函数执行完毕后, 控制权将返回到调用函数。函数 `main()` 执行完毕后, 控制权将返回到操作系统。



## 6.5 确定变量的作用域

变量都有作用域，作用域决定了变量在程序中的存活时间以及在什么地方可以访问它。在语句块中声明的变量的作用域为该语句块，只能在该语句块中访问它，离开该语句块后，它将不复存在。全局变量的作用域为全局，在程序的任何地方都可用。

### 6.5.1 局部变量

不仅可以将变量传递给函数，还可以在函数体中声明变量。在函数体内声明的变量被称为局部变量，因为它们只在函数中存在。当函数返回时，局部变量不再可用，编译器对其进行标记，以便销毁。局部变量的定义方法与其他变量相同。传入函数的参数也可被视为局部变量，可以在函数体内使用它们，就像在函数体中定义了一样。程序清单 6.2 是一个使用参数和函数中定义的局部变量的例子。

程序清单 6.2 局部变量和参数的用法

```
1: #include <iostream>
2:
3: float Convert(float);
4: int main()
5: {
6:     using namespace std;
7:
8:     float TempFer;
9:     float TempCel;
10:
11:     cout << "Please enter the temperature in Fahrenheit: ";
12:     cin >> TempFer;
13:     TempCel = Convert(TempFer);
14:     cout << "\nHere's the temperature in Celsius: ";
15:     cout << TempCel << endl;
16:     return 0;
17: }
18:
19: float Convert(float TempFer)
20: {
21:     float TempCel;
22:     TempCel = ((TempFer - 32) * 5) / 9;
23:     return TempCel;
24: }
```

▼ 输出:

```
Please enter the temperature in Fahrenheit: 212

Here's the temperature in Celsius: 100

Please enter the temperature in Fahrenheit: 32

Here's the temperature in Celsius: 0

Please enter the temperature in Fahrenheit: 85

Here's the temperature in Celsius: 29.4444
```

▼ 分析:

在第 8 行和第 9 行声明了两个 float 变量，一个用于存储华氏温度，另一个用于存储摄氏温度。第 11 行要求用户输入一个华氏温度，第 13 行将这个值传递给函数 Convert()。第 13 行调用 Convert() 后，跳到函数 Convert() 的第 1 行（即第 21 行）执行，这里声明了一个局部变量，其名称也为 Tempcel。注意，这个局部变量与第 9 行的 TempCel 变量不是同一变量，它只在

函数 Convert() 内存在。作为参数传入的值 TempFer 也只是 main() 传递的变量的一个局部副本。

在该函数中，可以给参数和局部变量以其他任何名称，程序的功能将不变。例如，将 TempFer 改为 FerTemp 或将 TempCel 改为 CelTemp 都是完全合法的，函数的功能将不变。如果读者修改这些名称，并重新编译程序，将发现这是可行的。

在函数中，将参数 TempFer 的值减去 32，然后乘以 5 并除以 9，再将结果赋给局部变量 TempCel，然后将这个值作为函数的返回值返回。在第 13 行，这个返回值被赋给函数 main() 中的变量 TempCel，第 15 行打印它。

上述输出表明，该程序运行了 3 次。第 1 次的输入值为 212，这是水的沸点的华氏温度，转换后的摄氏温度为 100。第 2 次输入水的冰点温度进行测试。第 3 次输入了一个随机数，转换结果为小数。

## 6.5.2 作用域为语句块的局部变量

可以在函数的任何地方定义变量，而不仅限于函数开头。变量的作用域为定义它的语句所在的语句块。也就是说，如果在函数中的一对大括号内定义了一个变量，则该变量只在该语句块中可用。程序清单 6.3 说明了这一点。

程序清单 6.3 作用域为语句块的变量

---

```

1: // Listing 6.3 - demonstrates variables
2: // scoped within a block
3:
4: #include <iostream>
5:
6: void myFunc();
7:
8: int main()
9: {
10:     int x = 5;
11:     std::cout << "\nIn main x is: " << x;
12:
13:     myFunc();
14:
15:     std::cout << "\nBack in main, x is: " << x;
16:     return 0;
17: }
18:
19: void myFunc()
20: {
21:     int x = 8;
22:     std::cout << "\nIn myFunc, local x: " << x << std::endl;
23:
24:     {
25:         std::cout << "\nIn block in myFunc, x is: " << x;
26:
27:         int x = 9;
28:
29:         std::cout << "\nVery local x: " << x;
30:     }
31:
32:     std::cout << "\nOut of block, in myFunc, x: " << x << std::endl;
33: }

```

---

### ▼ 输出:

```

In main x is: 5
In myFunc, local x: 8

In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8
Back in main, x is: 5

```

---

### ▼ 分析:

该程序首先在第 10 行对 `main()` 中的局部变量 `x` 进行初始化。第 11 行的打印结果证明 `x` 已被初始化为 5。在第 13 行,调用了函数 `MyFunc()`。

在函数 `MyFunc()` 中,第 21 行将一个名称也为 `x` 的局部变量初始化为 8,第 22 行打印该变量的值。

第 24 行的左大括号是一个语句块的开始标记。第 24 行再次打印作用域为函数 `myFunc()` 的变量 `x`。在第 27 行,定义了一个名称也为 `x` 的新变量,但作用域为当前语句块;它被初始化为 9。第 29 行打印最新的变量 `x` 的值。语句块在第 30 行结束,此时离开了第 27 行定义的变量 `x` 的作用域,它不再可见。

第 32 行打印了变量 `x` 的值,这是函数 `myFunc()` 中第 21 行定义的 `x` 变量。该变量的值不受位于语句块中的第 27 行定义的 `x` 的影响,仍为 8。

在第 33 行,函数 `myFunc()` 结束,其局部变量 `x` 不可用。程序返回到第 14 行。在第 15 行打印第 10 行定义的局部变量 `x`,该变量不受 `myFunc()` 中定义的任何变量的影响。

显然,如果给这 3 个变量提供不同的名称,该程序将清晰得多。

## 6.6 参数是局部变量

传入函数的参数为函数中的局部变量。修改这些参数不会影响调用函数中的值。这被称为按值传递,这意味着将在函数中创建每个参数的局部副本。这些局部副本与其他局部变量一样,程序清单 6.4 说明了这一点。

程序清单 6.4 按值传递

```
1: // Listing 6.4 - demonstrates passing by value
2: #include <iostream>
3:
4: using namespace std;
5: void swap(int x, int y);
6:
7: int main()
8: {
9:     int x = 5, y = 10;
10:
11:     cout << "Main. Before swap, x: " << x << " y: " << y << endl;
12:     swap(x,y);
13:     cout << "Main. After swap, x: " << x << " y: " << y << endl;
14:     return 0;
15: }
16:
17: void swap (int x, int y)
18: {
19:     int temp;
20:
21:     cout << "Swap. Before swap, x: " << x << " y: " << y << endl;
22:
23:     temp = x;
24:     x = y;
25:     y = temp;
26:
27:     cout << "Swap. After swap, x: " << x << " y: " << y << endl;
28: }
```

### ▼ 输出:

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

### ▼ 分析:

该程序在函数 `main()` 中初始化两个变量,然后将它们传送给函数 `swap()`,后者交换两个变量的值。然而,返回函数 `main()` 后再查看它们时,它们并未改变!

第 9 行对变量进行初始化,第 11 行打印它们的值。第 12 行调用函数 `swap()` 并传递变量。

程序跳到 `swap()` 函数处执行。在该函数中,第 21 行再次打印两个变量的值,结果与 `main()` 中相同。在第 23~25 行,交换变量的值,第 27 行的打印对此进行验证。在 `swap()` 函数中,两变量的值确实被互换。

然后返回到 `main()` 函数中的第 13 行继续执行,但在这里,变量的值并没有交换。

读者可能明白了,传递给函数 `swap()` 的参数是按值传递的,这意味着将在 `swap()` 中创建参数的局部副本。在第 23~25 行,交换的是局部变量的值,但函数 `main()` 中的变量不受影响。

第 8 章将介绍另一种传递参数的方式,这种方式让您能够修改 `main()` 中的变量。

## 6.6.1 全局变量

在函数外面定义的变量的作用域为全局,在程序的任何函数(包括 `main()`)中都可用。

与全局变量同名的局部变量不会修改全局变量,但会隐藏它。如果函数中有一个与全局变量同名的局部变量,则在函数中使用该名称时,指的是局部变量而不是全局变量。程序清单 6.5 说明了这些概念。

程序清单 6.5 全局变量和局部变量

```
1: #include <iostream>
2: void myFunction();           // prototype
3:
4: int x = 5, y = 7;           // global variables
5: int main()
6: {
7:     using namespace std;
8:
9:     cout << "x from main: " << x << endl;
10:    cout << "y from main: " << y << endl << endl;
11:    myFunction();
12:    cout << "Back from myFunction!" << endl << endl;
13:    cout << "x from main: " << x << endl;
14:    cout << "y from main: " << y << endl;
15:    return 0;
16: }
17:
18: void myFunction()
19: {
20:     using std::cout;
21:
22:     int y = 10;
23:
24:     cout << "x from myFunction: " << x << std::endl;
25:     cout << "y from myFunction: " << y << std::endl << std::endl;
26: }
```

### ▼ 输出:

```
x from main: 5
y from main: 7
```

```
x from myFunction: 5
y from myFunction: 10
```

```
Back from myFunction!
```

```
x from main: 5
y from main: 7
```

### ▼ 分析:

这个简单程序说明了几个有关局部变量和全局变量的容易混淆的要点。在第 4 行, 声明了两个全局变量 `x` 和 `y`。全局变量 `x` 被初始化为 5, `y` 被初始化为 7。

在 `main()` 函数中的第 9 行和第 10 行, 将这些变量的值打印到控制台。在函数 `main()` 中, 没有定义名称为 `x` 或 `y` 局部变量; 由于 `x` 和 `y` 为全局变量, 因此它们在函数 `main()` 中可用。

第 11 行调用函数 `myFunction()`, 程序跳到第 18 行执行。第 22 行定义了一个局部变量 `y`, 并将其初始化为 10。在第 24 行, `myFunction()` 打印变量 `x` 的值。这使用的是全局变量 `x`, 就像在 `main()` 中一样。然而, 在第 25 行使用变量名 `y` 时, 使用的是局部变量 `y`, 它隐藏了同名的全局变量。

函数执行完毕后返回到 `main()`, 再次打印全局变量的值。注意, 给 `myFunction()` 的局部变量 `y` 赋值, 完全没有影响到全局变量 `y` 的值。

## 6.6.2 有关全局变量的注意事项

在 C++ 中, 全局变量是合法的, 但人们几乎不使用它。C++ 源于 C 语言, 在 C 语言中, 全局变量是一个危险但必不可少的工具。之所以必不可少, 是因为程序员经常需要让数据对很多函数来说可用, 而将数据作为参数在函数之间传递非常麻烦, 尤其当调用序列中的众多函数接受参数只是为了将其传递给其他函数时。

全局变量之所以危险, 是因为它们是共享的数据, 一个函数可能以另一个函数看不到的方式修改全局变量, 这会导致难以发现的错误。

## 6.7 创建函数语句时的考虑因素

对于函数体可包含的语句数量和类型几乎没有任何限制。虽然在函数中不能定义其他函数, 但可以调用其他函数, 几乎在所有的 C++ 程序中, `main()` 所做的就是调用其他函数。函数甚至可以调用自身, 这将在稍后有关递归的一节讨论。

虽然在 C++ 中, 对函数的长度没有限制, 但设计良好的函数通常较短。很多程序员都建议函数长度不超过一屏, 这样可以同时看到整个函数。虽然这个经验规则经常被优秀程序员打破, 但函数越短越容易理解和维护。

每个函数都应执行单个易于理解的任务。如果函数很大, 应考虑能否将其分解为几个小任务。

## 6.8 再谈函数实参

任何合法的 C++ 表达式都可用作函数实参, 包括常量、数学和逻辑表达式以及返回一个值的函数。重要的是, 表达式的结果必须与函数期望的实参类型匹配。

甚至可将函数作为参数, 毕竟函数将返回一个值, 其类型与函数的返回类型相同。然而, 将函数作为参数可能使代码难以理解和调试。

例如, 假设有函数 `myDouble()`、`triple()`、`square()` 和 `cube()`, 其中每个函数都返回一个值, 则可以编写这样的代码:

```
Answer = (myDouble(triple(square(cube(myValue)))));
```

可以以两种方式来看待上述语句。首先, 可以认为函数 `myDouble()` 将函数 `triple()` 作为参数。而 `triple()` 将函数 `square()` 作为参数, 后者又将函数 `cube()` 作为参数。函数 `cube()` 将变量 `myValue` 作为参数。

按相反的方向看时, 该语句将变量 `myValue` 作为参数传给函数 `cube()`, `cube()` 的返回值被作为参数传递给函数 `square()`, `square()` 的返回值又被传递给 `triple()`, 而 `triple()` 的返回值又被传递给 `myDouble()`。最



后的结果被赋给变量 Answer。

很难确定上述的功能（是在平方之前还是之后乘以 3？），同时，如果答案不正确，也难以确定哪个函数有问题。

一种替代方式是，将每步的结果都赋给一个中间变量：

```
unsigned long myValue = 2;
unsigned long cubed   = cube(myValue);      // cubed = 8
unsigned long squared = square(cubed);      // squared = 64
unsigned long tripled = triple(squared);    // tripled = 192
unsigned long Answer  = myDouble(tripled);  // Answer = 384
```

这样，可以检查每个中间结果，且执行顺序很明确。

#### 警告

C++使得编写紧凑代码（如前述范例中将函数 cube( )、square( )、triple( )和 myDouble( )组合起来的代码）非常容易，但这并不意味着应该这样做。与使代码应尽可能紧凑相比，让代码易于阅读，进而易于维护是更好的选择。

## 6.9 再谈返回值

函数要么返回一个值，要么返回 void。void 告诉编译器，函数不返回任何值。

要从函数返回一个值，可在关键字 return 后面加上要返回的值。这可以是返回一个值的表达式。

例如：

```
return 5;                // returns a number
return (x > 5);           // returns the result of a comparison
return (MyFunction());   // returns the value returned by calling another
                           function
```

如果 myFunction( )函数返回一个值，则上述 return 语句都是合法的。如果 x 不大于 5，第 2 条语句将返回 false，否则返回 true。返回的是表达式的值：false 或 true，而不是 x 的值。

遇到关键字 return 时，其后的表达式将作为函数的返回值，并立即返回到调用函数，return 后的所有语句都不会被执行。

在同一个函数中可以有多条 return 语句，程序清单 6.6 演示了这一点。

程序清单 6.6 可以包含多条返回语句

```
1: // Listing 6.6 - demonstrates multiple return
2: // statements
3: #include <iostream>
4:
5: int Doubler(int AmountToDouble);
6:
7: int main()
8: {
9:     using std::cout;
10:
11:     int result = 0;
12:     int input;
13:
14:     cout << "Enter a number between 0 and 10,000 to double: ";
15:     std::cin >> input;
16:
17:     cout << "\nBefore doubler is called... ";
18:     cout << "\ninput: " << input << " doubled: " << result << "\n";
19:
20:     result = Doubler(input);
21:
22:     cout << "\nBack from Doubler...\n";
23:     cout << "\ninput: " << input << " doubled: " << result << "\n";
24:
25:     return 0;
26: }
27:
```

```
28: int Doubler(int original)
29: {
30:     if (original <= 10000)
31:         return original * 2;
32:     else
33:         return -1;
34:     std::cout << "You can't get here!\n";
35: }
```

### ▼ 输出:

```
Enter a number between 0 and 10,000 to double: 9000

Before doubler is called...
input: 9000 doubled: 0

Back from doubler...

input: 9000    doubled: 18000

Enter a number between 0 and 10,000 to double: 11000
Before doubler is called...
input: 11000 doubled: 0

Back from doubler...
input: 11000 doubled: -1
```

### ▼ 分析:

第 14 行和第 15 行要求用户输入一个数，第 17 行和第 18 行打印这个数和局部变量 `result`。第 20 行调用函数 `Doubler()`，并将 `input` 作为参数传递给它。结果被赋给局部变量 `result`，第 23 行打印这个值。

在函数 `Doubler()` 中的第 30 行，检测参数是否大于 10 000。如果不是，函数返回 `original` 的 2 倍，否则返回 -1 作为错误标记。

第 34 行的语句永远不会被执行。因为无论输入值是否大于 10 000，函数都将在第 31 行或第 33 行（到达第 34 行之前）返回。优秀的编译器将发出警告，指出该语句不可能被执行，优秀的程序员应删除它。

### FAQ

`int main()` 与 `void main()` 的区别何在？应使用哪一个？这两种方式我都用过，都正常，那么，为什么要使用 `int main(){return 0;}` 呢？

答：对于大多数编译器来说，两者都可行，但只有 `int main()` 符合 ANSI 标准。因此，只能保证 `int main()` 在将来仍可行。

两者的区别如下：`int main()` 向操作系统返回一个值。当程序结束时，返回值可被批处理程序或调用程序的程序捕获，并通过检查返回值（即退出编码）判断程序是否成功执行。

虽然本书的程序没有使用函数 `main()` 的返回值，但 ANSI 标准要求将函数 `main()` 的返回类型声明为 `int`，因此本书遵循这种规定。

## 6.10 默认参数

对每个在函数原型或定义中声明的变量，调用函数都必须为其传递一个值。传递的值必须是声明的类型。因此，如果像下面这样声明了一个函数：

```
long myFunction(int);
```

该函数必须接受一个 `int` 变量。如果函数定义与此不符或传递的不是整数，将出现编译错误。

对于这种规则一种例外是，函数原型声明了参数默认值。默认值是在没有提供参数值时使用的一个值。上述声明可以重写成这样：

```
long myFunction (int x = 50);
```

该原型指出：函数 `myFunction()` 接受一个 `int` 参数，并返回一个 `long` 值；如果没有提供参数，则使用默认值 50。由于函数原型中可以不包含参数名，因此上面的声明也可写成这样：

```
long myFunction (int = 50);
```

声明默认参数对函数定义没有影响。在上述函数的定义中，函数头如下：

```
long myFunction (int x)
```

如果调用该函数时没有提供参数，编译器将把 `x` 设置为默认值 50。在原型和函数头中，默认参数的名称可以不同，默认值是根据位置而不是参数名指定的。

可以给任何函数参数指定默认值，但有一条限制：如果某个参数没有默认值，它前面的所有参数都不得有默认值。

如果函数原型如下：

```
long myFunction (int Param1, int Param2, int Param3);
```

则仅当给 `Param3` 指定默认值后，才能给 `Param2` 指定默认值；仅当给 `Param2` 和 `Param3` 都指定了默认值后，才能给 `Param1` 指定默认值。程序清单 6.7 演示了默认值的用法。

#### 程序清单 6.7 默认参数值

---

```
1: // Listing 6.7 - demonstrates use
2: // of default parameter values
3: #include <iostream>
4:
5: int AreaCube(int length, int width = 25, int height = 1);
6:
7: int main()
8: {
9:     int length = 100;
10:    int width = 50;
11:    int height = 2;
12:    int area;
13:
14:    area = AreaCube(length, width, height);
15:    std::cout << "First area equals: " << area << "\n";
16:
17:    area = AreaCube(length, width);
18:    std::cout << "Second time area equals: " << area << "\n";
19:
20:    area = AreaCube(length);
21:    std::cout << "Third time area equals: " << area << "\n";
22:    return 0;
23: }
24:
25: int AreaCube(int length, int width, int height)
26: {
27:
28:     return (length * width * height);
29: }
```

---

#### ▼ 输出：

```
First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

#### ▼ 分析：

在第 5 行，函数 `AreaCube()` 的原型指定该函数接受 3 个 `int` 参数，其中最后两个有默认值。

该函数计算立方体的体积，立方体的尺寸是通过参数传入的。如果没有传递宽度，则宽度为 25，高度为 1。如果传入宽度但没有传入高度，则高度为 1。如果没有传递宽度，则不能传递高度。

在第 9~11 行，对 `length`、`width` 和 `height` 进行初始化。第 14 行将它们传递给函数 `AreaCube()`。第 15 行打印使用这些数值计算得到的结果。

然而继续执行第 17 行，这里再次调用了函数 `AreaCube()`，但没有指定高度。因此使用默认值，再

次计算面积并打印结果。

然后继续执行第 20 行，这次既没有指定宽度，也没有指定高度。该函数调用导致第 3 次调到第 25 行执行。函数使用默认值计算面积，然后返回到 `main()` 函数，在这里打印最后的值。

应该

请切记，函数参数在函数中的局部变量。

请切记，在一个函数中修改全局变量，将影响所有使用该变量的函数。

不应该

如果第二个参数没有默认值时，不要为第一个参数指定默认值。

别忘了，按值传递参数时不会影响调用函数中变量的值。

## 6.11 重载函数

C++ 允许创建多个名称相同的函数，这被称为函数重载。在这些同名函数的参数列表中，必须有不同的参数类型、参数个数或兼而有之。下面是一个例子：

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

函数 `myFunction()` 被重载了三次。前两个版本之间的区别在于参数类型不同，第三个版本与前两个版本之间的区别在于参数个数不同。

重载函数的返回类型可以相同，也可以不同。然而，不同的重载版本不能仅仅是返回类型不同，也就是说，它们还应接受不同的参数：

```
int myFunction (int);
void myFunction (int);           // illegal - as it differs only in return type
void myFunction (long);         // OK!
void myFunction (long, long);   // OK!
int myFunction (long, long);     // illegal - as it differs only in return type
int myFunction (long, int);      // OK!
int myFunction (int, long);      // OK!
```

正如读者看到的，函数的重载版本必须有独特的特征标，即接受的参数类型不同。

**注意**

如果两个函数的函数名和参数列表都相同，但返回类型不同，将导致编译错误。要修改返回类型，必须同时修改特征标（名称和/或参数列表）。

函数重载也叫做函数多态（polymorphism）。多态指的是多种形态。

函数多态指的是可以对函数进行重载使其有多种含义。通过修改参数的个数或类型，可以让多个函数使用相同的名称，进而根据指定的参数，调用与之匹配的函数。这让您能够只使用同一个函数来计算 `int` 值、`double` 值或其他类型值的平均值，而不必为每个函数使用不同的函数名，如 `AverageInts()`、`AverageDoubles()` 等。

假设要编写一个将输入值翻倍的函数，可能希望能够将 `int`、`long`、`float` 或 `double` 参数传递给它。如果不使用函数重载，必须创建 4 个函数名：

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

使用函数重载，可以做如下声明：

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

这更容易理解和使用。您不必考虑应该调用哪个函数，而只需传递一个变量就自动调用正确的函数。程序清单 6.8 演示了函数重载的用途。

程序清单 6.8 函数多态

```
1: // Listing 6.8 - demonstrates
2: // function polymorphism
3: #include <iostream>
4:
5: int Double(int);
6: long Double(long);
7: float Double(float);
8: double Double(double);
9:
10: using namespace std;
11:
12: int main()
13: {
14:     int    myInt = 6500;
15:     long   myLong = 65000;
16:     float  myFloat = 6.5F;
17:     double myDouble = 6.5e20;
18:
19:     int    doubledInt;
20:     long   doubledLong;
21:     float  doubledFloat;
22:     double doubledDouble;
23:
24:     cout << "myInt: " << myInt << "\n";
25:     cout << "myLong: " << myLong << "\n";
26:     cout << "myFloat: " << myFloat << "\n";
27:     cout << "myDouble: " << myDouble << "\n";
28:
29:     doubledInt = Double(myInt);
30:     doubledLong = Double(myLong);
31:     doubledFloat = Double(myFloat);
32:     doubledDouble = Double(myDouble);
33:
34:     cout << "doubledInt: " << doubledInt << "\n";
35:     cout << "doubledLong: " << doubledLong << "\n";
36:     cout << "doubledFloat: " << doubledFloat << "\n";
37:     cout << "doubledDouble: " << doubledDouble << "\n";
38:
39:     return 0;
40: }
41:
42: int Double(int original)
43: {
44:     cout << "In Double(int)\n";
45:     return 2 * original;
46: }
47:
48: long Double(long original)
49: {
50:     cout << "In Double(long)\n";
51:     return 2 * original;
52: }
53:
54: float Double(float original)
55: {
56:     cout << "In Double(float)\n";
57:     return 2 * original;
58: }
59:
60: double Double(double original)
61: {
62:     cout << "In Double(double)\n";
63:     return 2 * original;
64: }
```



### ▼ 输出:

```
myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13
DoubledDouble: 1.3e+21
```

### ▼ 分析:

函数 `Double()` 被重载成能够接受 `int`、`long`、`float` 或 `double` 作为参数。函数原型位于第 5~8 行, 定义位于第 42~64 行。

在这个例子中, 第 10 行使用了语句 `using namespace std;`, 它不在任何函数中, 这使得该声明在整个文件中有效, 因此该名称空间在该文件中声明的所有函数中都可用。

在主程序中, 声明了 8 个局部变量。在第 14~17 行, 对其中的 4 个变量进行了初始化; 在第 29~32 行, 将这 4 个变量分别传递给 `Double()` 函数, 并将结果分别赋给余下的 4 个变量。调用 `Double()` 函数时, 并没有指定要调用哪个版本, 只需传递一个参数就将调用正确的版本。

编译器对参数进行检查, 以决定选择 4 个 `Double()` 函数中的哪一个。输出表明, 4 个函数被依次调用, 与预期相同。

## 6.12 函数特有的主题

函数对编程非常重要, 一些函数特有的主题对解决不同寻常的问题可能会有所帮助。如果使用得当, 内联函数有助于提高性能。函数递归是奇妙、高深的编程技巧之一, 它常常能使其他方法无法解决的问题迎刃而解。

### 6.12.1 内联函数

当您定义函数时, 编译器通常会在内存中创建一组指令。当您调用函数时, 程序将跳到这些指令处执行, 当函数返回时, 程序跳到调用函数的下一行继续执行。如果调用同一个函数 10 次, 则每次程序都将跳到同一组指令处。这意味着内存中只有一个函数副本, 而不是 10 个。

跳入和跳出函数都存在一些影响性能的开销。有些函数非常小, 只有一两行代码, 如果可避免程序仅为执行一两条指令而进行跳转, 可能提高程序的效率。程序员说到效率时, 通常指的是速度。如果能够避免调用函数, 程序的运行将更快。

如果声明函数时使用了关键字 `inline`, 编译器将不会创建函数, 而直接将内联函数的代码复制到调用函数中。这样, 便不需要进行跳转, 就像函数的语句被写到调用函数中一样。

注意, 内联函数可能使成本大大增加。如果函数被调用 10 次, 内联代码将被复制到这 10 个函数调用处。这样, 在速度方面的改善可能微不足道, 而可执行程序变大了, 因此程序的速度可能更慢。

事实上, 现在优化编译器在做出这种决策方面比您更出色, 因此除非函数只有一两条语句, 否则不要将函数声明为内联的。如果没有把握, 则不应使用关键字 `inline`。如果程序员指定的内联函数太大, 导致生成的可执行文件急剧膨胀, 有些编译器可能不会将其作为内联函数。

**注意**

性能优化是一个复杂的问题，大多数程序员在没有帮助的情况下，并不擅长找出程序中存在性能问题的地方。

优化性能的正确方式是，使用剖析程序研究应用程序的行为。剖析程序可提供各种统计信息，从执行特定函数花费的时间到函数被调用多少次。这些统计信息有助于程序员将精力放在值得关注的代码上，而不是仅凭感觉去优化代码，导致收获极其有限。

鉴于这种原因，与对哪种情况下运行速度会更快或更慢进行猜测，进而编写出难以理解的代码相比，编写清晰易懂的代码总是一种更好的选择。这是因为提高易于理解的代码的速度更容易。

程序清单 6.9 演示了一个内联函数。

程序清单 6.9 内联函数

```
1: // Listing 6.9 - demonstrates inline functions
2: #include <iostream>
3:
4: inline int Double(int);
5:
6: int main()
7: {
8:     int target;
9:     using std::cout;
10:    using std::cin;
11:    using std::endl;
12:
13:    cout << "Enter a number to work with: ";
14:    cin >> target;
15:    cout << "\n";
16:
17:    target = Double(target);
18:    cout << "Target: " << target << endl;
19:
20:    target = Double(target);
21:    cout << "Target: " << target << endl;
22:
23:    target = Double(target);
24:    cout << "Target: " << target << endl;
25:    return 0;
26: }
27:
28: int Double(int target)
29: {
30:     return 2*target;
31: }
```

**▼ 输出:**

Enter a number to work with: 20

Target: 40

Target: 80

Target: 160

**▼ 分析:**

在第 4 行，函数 `MyDouble()` 被声明为一个内联函数，它接受一个参数，返回一个 `int` 值。除在返回类型前加上关键字 `inline` 外，该声明与其他原型类似。

编译上述代码时，就像在输入下述语句的每个地方输入了 `target = 2 * target;` 一样：

```
target = Double(target);
```

程序执行时，指令已经就位，并被编译为 `.obj` 文件。这避免了在执行程序时跳转和返回的开销，但代价是可执行程序更大。

**注意**

关键字 `inline` 是一种提示，它告诉编译器，您希望函数是内联的。编译器可以忽略这种提示，创建真正的函数调用。

## 6.12.2 递归

函数可以调用自身，这被称为递归。递归可以是直接的或间接的。直接递归是指函数调用自身，而间接递归是指函数调用了另一个函数，而后者又调用了它。

使用递归能轻松地解决一些问题，在这些问题中，通常需要对数据进行某种处理，然后对结果进行相同的处理。两种递归（直接和间接）都有两种可能的结果，一种是最终结束递归并得出答案，另一种是无休止地递归下去，导致运行阶段错误。

需要注意的是，函数调用自身时，系统将在内存创建该函数的一个新副本。新副本和前一个副本中的局部变量彼此独立，它们之间不会彼此直接影响，就像程序清单 6.3 表明的，`main()` 中的局部变量不能影响被调用的函数中的局部变量一样。

为演示如何使用递归解决问题，请考虑下面的 Fibonacci 数列：

1, 1, 2, 3, 5, 8, 13, 21, 34...

这个数列中，从第 3 个数开始，每个数都是它前面两个数之和。一个可能的 Fibonacci 问题是，计算该数列中第 12 个数的值。

为解决这个问题，必须仔细分析这个数列。前两个数都为 1，以后的每个数都是其前两个数的和。因此，第 7 个数是第 5 个和第 6 个数的和。推而广之，如果  $n$  大于 2，则第  $n$  个数是第  $n-1$  个数和第  $n-2$  个数之和。

递归函数需要一个结束条件。必须有某件事情导致程序停止递归，否则递归将没完没了。在 Fibonacci 数列中， $n < 3$  是结束条件（也就是说，当  $n < 3$  时，程序可以结束对问题的处理）。

算法是一组解决问题时遵循的步骤。对于 Fibonacci 数列，一种算法如下：

1. 要求用户输入数列中的一个位置；

2. 调用 `fib()` 函数，并将用户输入的值传给它；

3. 函数 `fib()` 检查参数  $n$ ，如果  $n < 3$ ，则返回 1；否则 `fib()` 递归调用自身，并将  $n-2$  传递给它，然后再次调用自身，并将  $n-1$  传递给它，最后，返回这两次调用的返回值之和。

如果调用 `fib(1)`，将返回 1；如果调用 `fib(2)`，也将返回 1；如果调用 `fib(3)`，将返回函数调用 `fib(1)` 和 `fib(2)` 的返回值之和。由于 `fib(1)` 和 `fib(2)` 均返回 1，因此 `fib(3)` 返回 2 (1+1)。

如果调用 `fib(4)`，将返回函数调用 `fib(3)` 和 `fib(2)` 的返回值之和。已知 `fib(3)` 返回 2（通过调用 `fib(2)` 和 `fib(1)`），`fib(2)` 返回 1，因此 `fib(4)` 返回 3，这是数列中的第 4 个数。

再进一步，如果调用 `fib(5)`，将返回函数调用 `fib(4)` 和 `fib(3)` 的返回值之和。已知 `fib(4)` 返回 3，`fib(3)` 返回 2，因此返回的和为 5。

这种方法并不是解决该问题最有效的方法（对于 `fib(20)`，将调用函数 `fib()` 13 529 次），但确实可行。请注意，如果用户输入的数过大，内存将被耗尽。每次调用 `fib()` 时，都需要分配内存；返回时，内存被释放。使用递归时，将不断分配内存，系统内存将很快被耗尽。程序清单 6.10 实现了函数 `fib()`。

**警告**

运行程序清单 6.10 时，输入的数越大，导致的递归函数调用越多，进而消耗的内存越多。

### 程序清单 6.10 使用 Fibonacci 数列演示递归

```
1: // Fibonacci series using recursion
2: #include <iostream>
3: int fib (int n);
4:
```

```

5: int main()
6: {
7:
8:     int n, answer;
9:     std::cout << "Enter number to find: ";
10:    std::cin >> n;
11:
12:    std::cout << "\n\n";
13:
14:    answer = fib(n);
15:
16:    std::cout << answer << " is the " << n;
17:    std::cout << "th Fibonacci number\n";
18:    return 0;
19: }
20:
21: int fib (int n)
22: {
23:     std::cout << "Processing fib(" << n << ")... ";
24:
25:     if (n < 3 )
26:     {
27:         std::cout << "Return 1!\n";
28:         return (1);
29:     }
30:     else
31:     {
32:         std::cout << "Call fib(" << n-2 << ") ";
33:         std::cout << "and fib(" << n-1 << ").\n";
34:         return( fib(n-2) + fib(n-1));
35:     }
36: }

```

### ▼ 输出:

Enter number to find: 6

```

Processing fib(6)... Call fib(4) and fib(5).
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(5)... Call fib(3) and fib(4).
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
8 is the 6th Fibonacci number

```

### 注意

有些编译器不能处理在 cout 语句中使用的运算符的情况。如果编译器发出警告，指出第 32 行有问题，请使用括号将减法运算括起，使第 32 行和第 33 行变成如下所示：

```

std::cout << "Call fib(" << (n-2) << ") ";
std::cout << "and fib(" << (n-1) << ").\n";

```

### ▼ 分析:

这个程序在第 9 行要求用户指定要计算第几个数的值，并将其赋给 n。然后调用函数 fib()，并将 n 作为参数传递给它。程序跳到 fib() 函数处执行，在该函数中，第 23 行打印参数。

第 25 行检测参数 n 是否小于 3，如果是，fib() 返回 1，否则，调用 fib(n-2) 和 fib(n-1)，并返回这两个函数调用的返回值之和。

对 fib() 的递归调用结束前，不会返回这些值。为描绘该程序，可将 fib() 调用不断细分，直到

fib( )调用能够直接返回一个值为止。只有调用 fib(2)和 fib(1)可直接返回一个值。这些值向上传递给调用函数，后者将它们相加，得到自己的返回值，然后返回。图 6.4 和图 6.5 说明了对 fib( )的递归调用过程。

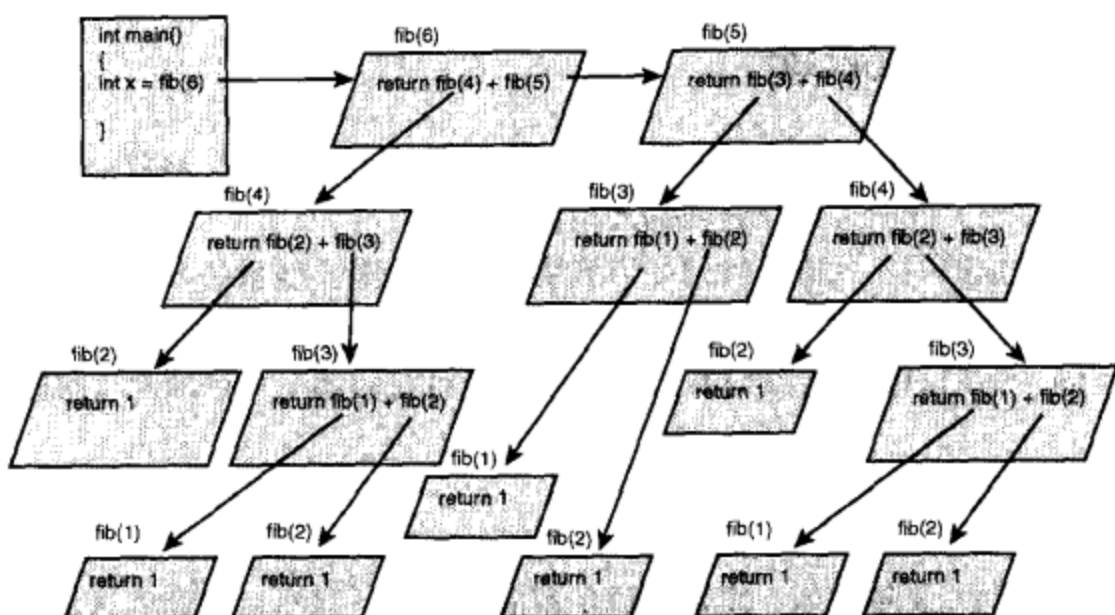


图 6.4 使用递归

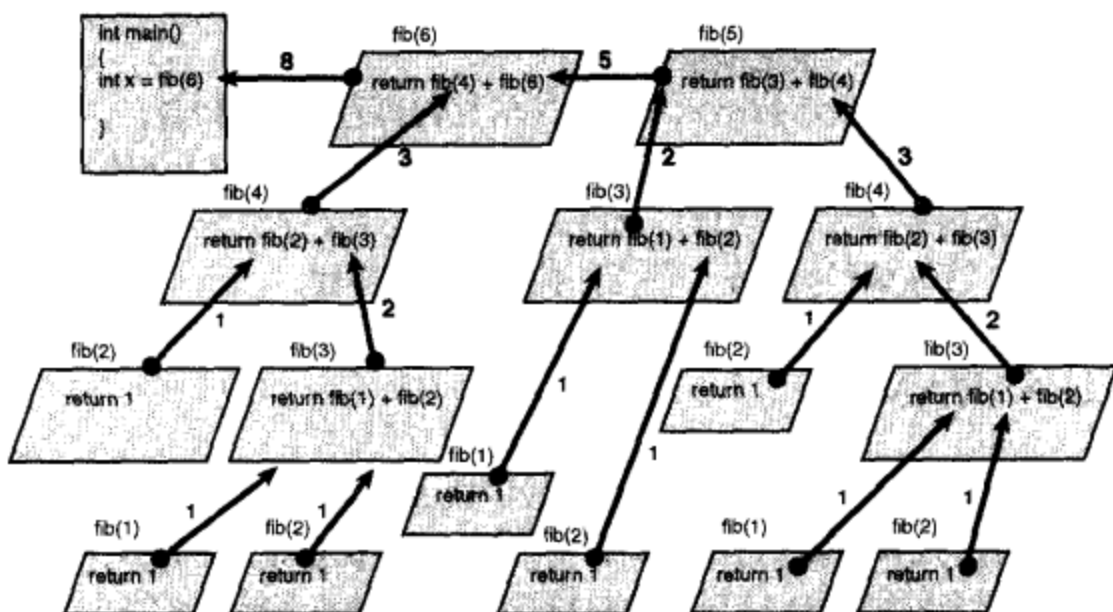


图 6.5 从递归调用返回

在这个例子中，n 为 6，因此 main() 调用 fib(6)。程序跳到 fib() 函数处执行。第 25 行检测 n 是否小于 3，结果为假，因此 fib(6) 在第 34 行调用 fib(5) 和 fib(4)，并返回它们的返回值之和。第 34 行的代码如下：

```
return( fib(n-2) + fib(n-1));
```

该返回语句调用 fib(4) (由于  $n=6$ , 因此 fib( $n-2$ ) 就是 fib(4)) 和 fib(5) (fib( $n-1$ )). 然后, 函数调用 fib(6) 处于等待状态, 直到这些调用都返回一个值。这些函数调用都返回一个值后, 函数调用 fib(6) 便可以返回这两个值的和。

由于函数 fib(5) 的参数不小于 3，因此再次 fib()，这次使用参数 4 和 3。然后，fib(4) 将调用 fib(3) 和 fib(2)。

前面的输出跟踪了这些调用和返回值。请编译、链接和运行该程序，依次输入 1、2、3，最后输入 6，并仔细观察输出。

这是尝试使用调试器的绝好时机。在第 21 行设一个断点，然后跟踪每次 fib() 调用，每递归调用 fib() 时，观察 n 值的变化。

在 C++ 编程中，并不经常使用递归，但对于某些问题，它是一个方便而有力的工具。



## 注意

递归是高级编程中比较棘手的部分。之所以在这里介绍它，旨在让读者了解其基本原理。如果读者不能完全理解所有的细节，也不必担心。

## 6.13 函数的工作原理

调用函数时，跳转到被调用的函数处，接着传入参数并执行函数体。函数结束时，如果返回类型不为 void，将返回一个值，同时控制权返回到调用函数。

这种任务是如何完成的呢？代码是如何知道应跳转到何处呢？变量传入后被存储在哪里？在函数体中声明变量的又如何呢？返回值如何被传回？代码如何知道该返回到哪里？

为回答这些问题，必须简要地介绍计算机内存。读者可在阅读下一章前再次阅读本节。

### 6.13.1 抽象层次

新程序员遇到的一个主要难点是，需要理解众多抽象层次。当然，计算机只是一种电子机器。它们没有窗口和菜单的概念，也没有程序和指令的概念，甚至没有 0 和 1 的概念。实际发生的一切就是在集成电路的各个不同位置测量电压。即使这样描述仍然是一种抽象：电本身是表示亚原子粒子行为的一个抽象概念，而这些粒子本身也是抽象概念。

很少有程序员会不厌其烦地去了解比 RAM 中的值更低层的细节。毕竟，并不需要了解粒子物理学，就可以开汽车、烤面包或打棒球，同样不必了解计算机的电子构造也可以编程。

然而，确实需要了解内存是如何组织的。如果在创建变量时对于变量位于何处以及值如何在函数间传递没有清楚的了解，那么编程仍然是一件无法控制的神秘事情。

### 6.13.2 划分 RAM

程序启动时，操作系统（如 DOS、UNIX/Linux 或微软 Windows）将依据编译器的需求设置各种内存区域。作为 C++ 程序员，经常需要关心的是全局名称空间、自由存储、寄存器、代码空间和堆栈。

全局变量都放在全局名称空间中。全局名称空间和自由存储将在接下来的几章详细介绍，这里将重点放在寄存器、代码空间和堆栈上。

寄存器是 CPU 中的一个特殊存储区域。它们负责进行 CPU 内部事务处理。寄存器的具体工作原理超出了本书的范围，但读者应该知道的是在任意给定时刻指向下一行代码的寄存器组。这些寄存器被统称为指令指针。指令指针的任务是跟踪接下来将执行哪行代码。

代码本身位于代码空间中，后者是分配用于存储程序中指令的二进制形式的那部分内存。每行源代码都被转换为一系列指令，每条指令都位于内存中的某个地址处。指令指针中存放了接下来要执行的指令的地址。图 6.6 说明了这种概念。

堆栈是为程序分配的一块特殊内存区域，用来存储程序中每个函数所需的数据。之所以称为堆栈，是因为它是后进先出的，就像咖啡馆中堆放盘子的架子一样，如图 6.7 所示。

后进先出意味着最后被加入到堆栈中的东西首先被取出。这不同于大多数队列，后者是先进先出的，就像剧院售票处前的长队，谁先来谁先走。堆栈像一堆硬币，如果在桌上堆放 10 枚硬币，然后再取走一些，则最后放进去的 3 枚硬币就是最先被取走的 3 枚硬币。

数据被压入堆栈后，堆栈将增大；数据被弹出堆栈后，堆栈将缩小。像放盘子的架子一样，上面的盘子没拿走之前，无法拿走下面的盘子。

盘子架是一种常见的类比。就当前情形而言，这种类比是正确的，但从基本原理上说，这是错误

的。一种更准确的类比是，堆栈像一排上下对齐的方盒子。栈顶是堆栈指针（另一个寄存器）指向的方盒子。

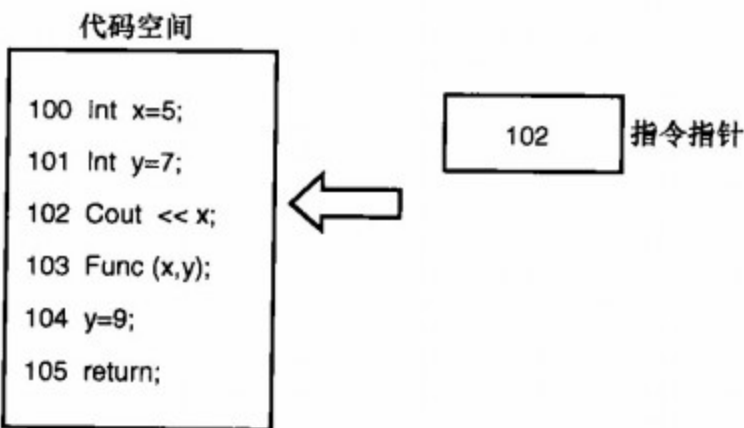


图 6.6 指令指针

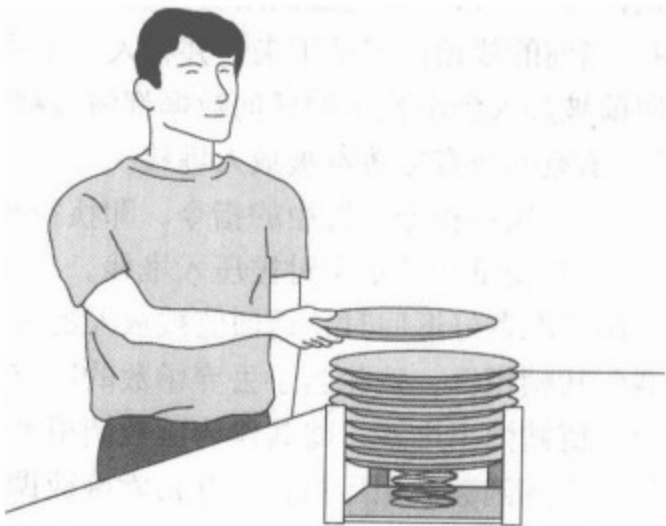


图 6.7 堆栈

每个方盒子都有一个顺序排列的地址，其中有一个地址保存在堆栈指针寄存器中，这个地址被称为栈顶，其下的所有数据都被认为是存放于堆栈中，其上的所有数据都被认为在堆栈外，因而无效。图 6.8 说明了这一点。

数据被压入堆栈时，放在堆栈指针上面的方盒子中，然后堆栈指针上移，指向新数据。当数据弹出堆栈时，堆栈指针沿堆栈下移。图 6.9 说明了这种规则。

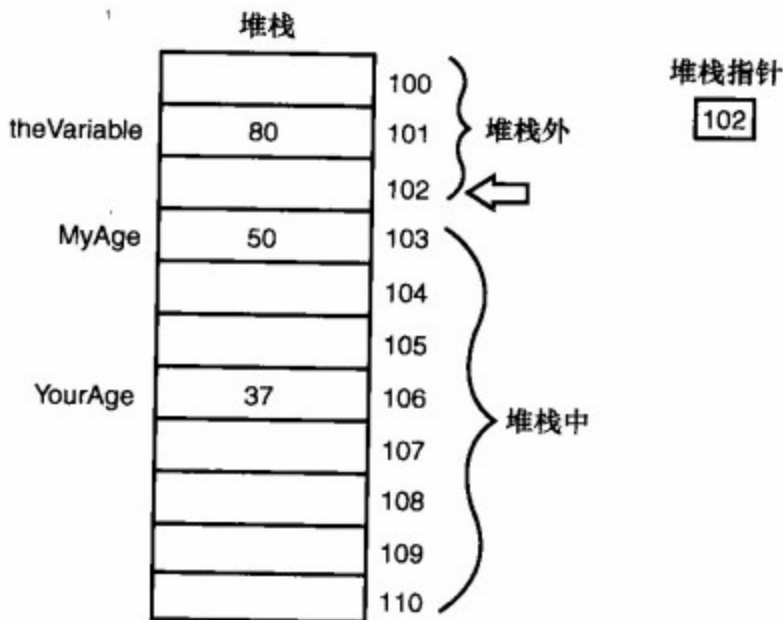


图 6.8 堆栈指针

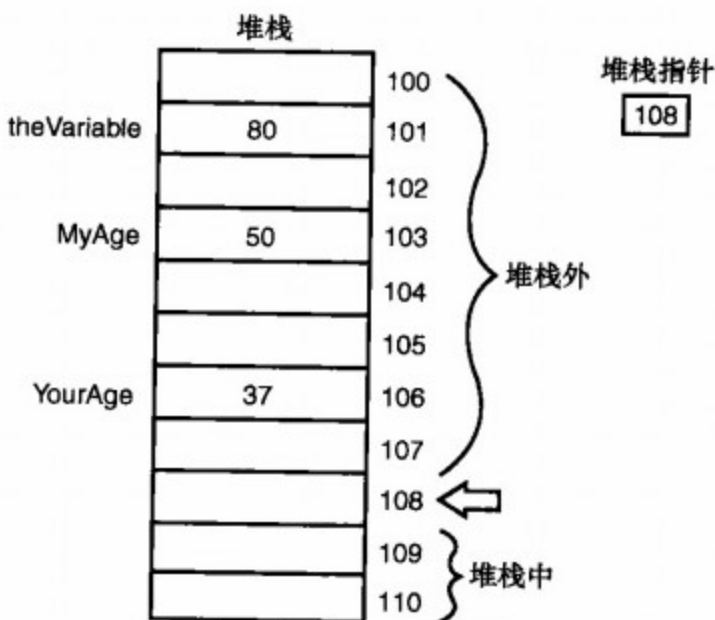


图 6.9 移动堆栈指针

位于堆栈指针以上（栈外）的数据在任何时候都可能变化，也可能不变，这些值被称为垃圾，因为它们的价值是不可靠的。

### 6.13.3 堆栈和函数

以下是程序跳转到函数处执行时发生的大致情况（具体细节随操作系统和编译器而异）。

1. 指令指针中的地址值增加 1，指向函数调用后的下一条指令。这个地址随后被放入堆栈，它是函数返回时的返回地址。
2. 在堆栈中为声明的返回值类型分配空间。如果在 int 变量占两字节的系统上，返回类型被声明为 int，堆栈再增加两个字节，但在这个两字节中不存放任何值（这意味着这两个字节中的“垃圾”保

持不变，直到本地变量被初始化)。

3. 被调函数的地址存储在为此而分配的一块特殊内存区域中，这个地址被加载到指令指针中，这样将执行的下一条指令为被调用的函数。

4. 当前的栈顶被记录下来，并存入一个称为栈帧 (stack frame) 的特殊指针中。从现在开始到函数返回前被加入到堆栈中的任何数据都将被视为函数的局部数据。

5. 函数的所有参数都被放入堆栈。

6. 现在执行指令指针中的指令，即执行函数的第一条指令。

7. 局部变量在被定义时被压入堆栈。

当函数准备好返回时，返回值被放入第 2 步预留的堆栈区域中。随后不断对堆栈执行弹出操作，直接遇到栈帧指针，这相当于丢弃函数的所有局部变量和参数。

返回值被弹出堆栈，将其作为函数调用本身的价值。然后检索第 1 步存储的地址，将其放入指令指针。程序回到函数调用后执行，并检索函数调用的返回值。

上述过程中的一些细节可能随计算机操作系统、编译器和处理器而异，但基本思想是一致的。一般而言，调用函数时，返回地址和参数将被压入堆栈。在函数执行期间，局部变量也被压入堆栈。函数返回时，这些值都从堆栈中弹出，从而被删除。

第 8 章将介绍内存的其他区域，这些区域用来存储那些生命周期比函数更长的数据。

## 6.14 总结

本章介绍了函数。函数实际上可以被看作是输入参数并返回一个值的子程序。每个 C++ 程序都从 `main()` 函数开始执行，而 `main()` 函数可以调用其他函数。

函数是使用函数原型声明的，原型描述了返回值、函数名和参数类型。函数可声明为内联的。在函数原型中，还可以为一个或多个参数声明默认值。

函数定义必须在返回类型、函数名和参数列表方面与原型一致。通过改变参数数目或类型，可以重载函数。编译器将根据参数列表找到正确的函数。

在函数中声明的变量以及传入函数的参数的作用域都是其声明所在的语句块。按值传递的参数只是副本，不会影响调用函数中变量的值。

## 6.15 问与答

问：为什么不将所有变量都声明为全局的？

答：在编程中，一度是这样做的。但随着程序越来越复杂，这样做就使得很难发现程序中的错误，因为任何一个函数都可能破坏数据——在程序中的任何地方都可以修改全局变量的值。多年的编程经验使程序员们相信，数据应该尽量局部化，可修改数据的地方也应尽可能小。

问：什么情况下应在函数原型中使用关键字 `inline`？

答：如果函数很小，只有一两行，且不会在程序中的很多地方调用它，则可以将其声明为内联的。

问：为什么函数参数值的变化不会反映到调用函数中？

答：传递给函数的参数是按值传递的。这意味着函数中的参数只是原始值的一个副本。这种概念在 6.13 节中进行了详细介绍。

问：如果参数是按值传递的，要将参数值的变化反映到调用函数中该如何办？

答：第 8 章将讨论指针，而第 9 章将讨论引用。使用指针和引用可解决这种问题，同时还可突破函数只能返回一个值的限制。

问：如果有以下两个函数，将发生什么情况？

```
int Area (int width, int length = 1); int Area (int size);
```

它们是重载吗？这里虽然参数个数不同，但第一个函数指定了默认值。

答：这两个声明可以通过编译，但使用一个参数来调用函数 Area 时，将发生错误：无法判断应使用 Area(int, int)还是 Area(int)。

## 6.16 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 6.16.1 测验

1. 函数原型和函数定义之间有何区别？
2. 参数的名称在函数原型、函数定义和函数调用中必须一致吗？
3. 如果函数不返回任何值，如何声明它？
4. 如果没有声明返回值，默认返回类型是什么？
5. 什么是局部变量？
6. 什么是作用域？
7. 什么是递归？
8. 什么时候应该使用全局变量？
9. 什么是函数重载？

### 6.16.2 练习

1. 编写一个名为 Perimeter() 的函数的原型，它接受两个 unsigned short int 参数，返回类型为 unsigned long int。

2. 编写练习 1 中函数 Perimeter() 的定义。两个参数表示矩形的长和宽，函数返回周长（长与宽之和的两倍）。

3. 查错：下述代码中的函数有什么错误？

```
#include <iostream>
void myFunc(unsigned short int x);
int main()
{
    unsigned short int x, y;
    y = myFunc(int);
    std::cout << "x: " << x << " y: " << y << "\n";
    return 0;
}

void myFunc(unsigned short int x)
{
    return (4*x);
}
```

4. 查错：下述代码中的函数有什么错误？

```
#include <iostream>
int myFunc(unsigned short int x);
int main()
{
    unsigned short int x, y;
    x = 7;
    y = myFunc(x);
    std::cout << "x: " << x << " y: " << y << "\n";
    return 0;
}

int myFunc(unsigned short int x);
{
    return (4*x);
}
```

5. 编写一个函数，它接受两个 unsigned short 参数，返回第 1 个数与第 2 个数的商。如果第 2 个数为 0，则不执行除法，并返回-1。

6. 编写一个程序，请用户输入两个数，然后调用练习 5 的函数。打印结果；如果返回值为-1，打印错误消息。

7. 编写一个程序，要求用户输入一个数和一个指数。编写一个递归函数，接受这两个数作为参数，并计算幂运算结果。例如，如果数为 2，指数为 4，该函数返回 16。





## 第 7 章

# 控制程序流程

程序的大部分工作是通过分支和循环完成的。第 5 章介绍了如何使用 if 语句实现分支。

在本章中，您将学习：

- 什么是循环？如何使用它们
- 如何创建各种循环
- 深度嵌套 if...else 语句的替代品

## 7.1 循环

很多编程问题的解决是通过对相同的数据进行重复操作完成的。完成这种功能的两种方法是递归（第 6 章讨论过）和迭代。迭代意味着重复地做同样的工作。迭代的主要方法是循环。

### 7.1.1 循环的鼻祖：goto

在计算机科学发展的早期，程序糟糕、粗糙而简短。循环由标签、语句和跳转组成。

在 C++ 中，标签是后面跟冒号 (:) 的名称。标签放在合法 C++ 语句的左边，跳转是通过在关键字 goto 后面加上标签的名称实现的。程序清单 7.1 演示了这种原始的循环方式。

程序清单 7.1 使用关键字 goto 实现循环

```
1: // Listing 7.1
2: // Looping with goto
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     int counter = 0;          // initialize counter
9: loop:
10:    counter++;                // top of the loop
11:    cout << "counter: " << counter << endl;
12:    if (counter < 5)           // test the value
13:        goto loop;            // jump to the top
14:
15:    cout << "Complete. Counter: " << counter << endl;
16:    return 0;
17: }
```

#### ▼ 输出：

```
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

**▼ 分析:**

在第8行, counter 被初始化为0。第9行包含一个名为 loop 的标签, 标记循环的开始。该循环对 counter 进行递增, 然后在第11行打印其新值。第12行测试 counter 的值, 如果小于5, 则 if 语句为 true, 从而执行 goto 语句。这将导致程序跳转到第9行的 loop 标签处执行。程序不断循环, 直到 counter 的值为5后跳出循环, 打印最后的输出。

## 7.1.2 为何避免使用 goto 语句

作为一条原则, 程序员避免使用 goto 语句, 这是有原因的。goto 语句可以向前或向后跳转到源代码的任何位置。不加选择地使用 goto 语句会导致混乱、质量低下和无法阅读的程序, 这被称为“意大利面条”式代码。

**goto 语句**

要使用 goto 语句, 只需在 goto 后面加上标签名称, 这将导致程序无条件地跳转到标签处。

示例:

```
if (value > 10)
    goto end;
if (value < 10)
    goto end;
cout << "value is 10!";
end:
    cout << "done";
```

为避免使用 goto 语句, 引入了更高级、控制更严格的循环命令: for、while 和 do...while。

## 7.2 使用 while 循环

只要开始条件为真, while 循环将导致程序重复执行一段代码。在程序清单 7.1 所示的 goto 语句示例中, counter 被不断递增, 直到等于5为止。程序清单 7.2 使用 while 循环重新编写了该程序。

程序清单 7.2 while 循环

```
1: // Listing 7.2
2: // Looping with while
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     int counter = 0;          // initialize the condition
9:
10:    while(counter < 5)        // test condition still true
11:    {
12:        counter++;           // body of the loop
13:        cout << "counter: " << counter << endl;
14:    }
15:
16:    cout << "Complete. Counter: " << counter << endl;
17:    return 0;
18: }
```

**▼ 输出:**

```
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

### ▼ 分析:

这个简单程序说明有关 while 循环的基本知识。第 8 行创建了一个名为 counter 的 int 变量, 并将其初始化为 0。然而将其用作条件的一部分, 并对条件进行测试, 如果条件为真, 则执行 while 循环体。在这个例子中, 第 10 行测试的条件为 counter 是否小于 5。如果为真, 就执行循环体。第 11 行对 counter 进行递增, 第 12 行打印它的值。如果第 10 行的条件语句为假 (counter 不小于 5), 将跳过整个 while 循环体 (第 11~14 行), 进入第 15 行。

这里需要指出的是, 总是使用大括号将循环体括起是个不错的主意, 即使循环体只有一行代码。这样可避免这样一种常见的错误: 不小心在循环后面加上分号, 导致循环无休止地执行, 例如:

```
int counter = 0;
while ( counter < 5 );
    counter++;
```

在这个例子中, counter++ 永远不会被执行。

#### while 语句

while 语句的语法如下:

```
while ( condition )
    statement;
```

condition 可以是任何 C++ 表达式, statement 可以是任何合法的 C++ 语句或语句块。如果 condition 为 true, 将执行 statement, 然后再次测试 condition。这一过程将不断重复下去, 直到 condition 为 false 为止。此时, while 循环将终止, 接着执行 statement 后面的第一条语句。

示例:

```
// count to 10
int x = 0;
while (x < 10)
    cout << "X: " << x++;
```

### 7.2.1 更复杂的 while 语句

while 循环测试的条件可以与任何合法 C++ 表达式一样复杂, 这包括使用逻辑运算符 &&、|| 和 ! 组合而成的表达式。程序清单 7.3 列出了一个较复杂的 while 语句。

程序清单 7.3 复杂的 while 循环

```
1: // Listing 7.3
2: // Complex while statements
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     unsigned short small;
9:     unsigned long large;
10:    const unsigned short MAXSMALL=65535;
11:
12:    cout << "Enter a small number: ";
13:    cin >> small;
14:    cout << "Enter a large number: ";
15:    cin >> large;
16:
17:    cout << "small: " << small << "...";
18:
19:    // for each iteration, test two conditions
20:    while (small < large && small < MAXSMALL)
21:    {
22:        if (small % 5000 == 0) // write a dot every 5k lines
23:            cout << ".";
```

```

24:
25:     small++;
26:     large-=2;
27: }
28:
29:     cout << "\nSmall: " << small << " Large: " << large << endl;
30:     return 0;
31: }

```

**▼ 输出:**

```

Enter a small number: 2
Enter a large number: 100000
small: 2.....
Small: 33335 Large: 33334

```

**▼ 分析:**

该程序是一个游戏。用户输入两个数，一大一小。较小的数 (small) 将不断递增，而较大的数 (large) 每次减 2。游戏的目标是，用户猜出它们何时相等。

在第 12~15 行，读取用户输入的数字。第 20 行建立一个 while 循环，只要下面两个条件都满足，该循环将不断执行：

1. small 不大于 large。
2. small 小于 MAXSMALL。

在第 22 行，对 small 和 5000 执行求模运算。这并不会修改 small 的值，只是在 small 为 5000 的整数倍时返回 0。在这种情况下，将在屏幕上打印一个句点指出程序正在运行。第 25 行将 small 加 1，第 26 行将 large 减去 2。当 while 循环中两个条件的任何一个不满足时，循环结束，程序继续执行第 27 行的右大括号后面的语句。

**注意**

求模运算符 (%) 和复合条件在第 5 章介绍过。

## 7.2.2 continue 和 break 简介

有时候，需要在执行 while 循环体中所有语句之前返回到 while 循环的开头。continue 语句用于跳转到循环的开头。

而在另一些时候，需要在满足循环退出条件之前跳出循环。break 语句立即跳出 while 循环，继续执行右括号后的语句。

程序清单 7.4 演示了这些语句的用法。这次游戏更复杂，要求用户输入 small、large、skip 和 target。small 每次增加 1，large 每次减少 2。当 small 为 skip 的整数倍时，不将 large 减 2。small 大于 large 时，游戏结束。如果 large 刚好等于 target，则印一条消息，同时游戏结束。

用户的目标是，使游戏在 large 等于 target 时结束。

### 程序清单 7.4 break 和 continue

```

1: // Listing 7.4 - Demonstrates break and continue
2: #include <iostream>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     unsigned short small;
9:     unsigned long large;
10:    unsigned long skip;
11:    unsigned long target;
12:    const unsigned short MAXSMALL=65535;

```

```
13:
14:     cout << "Enter a small number: ";
15:     cin >> small;
16:     cout << "Enter a large number: ";
17:     cin >> large;
18:     cout << "Enter a skip number: ";
19:     cin >> skip;
20:     cout << "Enter a target number: ";
21:     cin >> target;
22:
23:     cout << "\n";
24:
25:     // set up 2 stop conditions for the loop
26:     while (small < large && small < MAXSMALL)
27:     {
28:         small++;
29:
30:         if (small % skip == 0) // skip the decrement?
31:         {
32:             cout << "skipping on " << small << endl;
33:             continue;
34:         }
35:
36:         if (large == target) // exact match for the target?
37:         {
38:             cout << "Target reached!";
39:             break;
40:         }
41:
42:         large-=2;
43:     } // end of while loop
44:
45:     cout << "\nSmall: " << small << " Large: " << large << endl;
46:     return 0;
47: }
```

#### ▼ 输出:

```
Enter a small number: 2
Enter a large number: 20
Enter a skip number: 4
Enter a target number: 6
```

```
skipping on 4
skipping on 8
```

```
Small: 10 Large: 8
```

#### ▼ 分析:

在这次游戏中，用户失败了。在 large 等于 target (6) 之前，small 已经比 large 大了。

第 26 行测试 while 循环条件。如果 small 比 large 小且 small 没有超过 short int 的最大允许值，则执行 while 循环体。

在第 30 行，对 small 和 skip 进行求模运算。如果 small 是 skip 的整数倍，将执行 continue 语句，跳转到第 26 行的循环开头。这将跳过对 target 和 large 是否相等的测试以及将 large 减 2 的操作。

在第 36 行，将 target 和 large 进行比较，如果它们相等，则用户获胜：打印一条消息后，到达并执行 break 语句，这将立刻跳出 while 循环，在第 44 行继续执行。

#### 注意

应慎用 continue 和 break 语句。它们的危险性仅次于 goto 语句，原因与 goto 语句相同。突然改变方向的程序难以理解，不加选择地使用 continue 和 break 语句，甚至会使很小的 while 循环难以理解。

需要中途跳出循环通常意味着没有使用合适的布尔条件设置循环终止条件。与使用 break 语句相比，在循环中使用 if 语句来跳过某些代码通常是一种更好的选择。



**continue 语句**

continue 语句导致 while、do...while 或 for 循环跳到循环开始处，有关使用 continue 语句的示例，请参阅程序清单 7.4。

**break 语句**

break 语句导致 while、do...while 或 for 循环立刻终止，跳到右大括号后面的语句处执行。

示例：

```
while (condition)
{
    if (condition2)
        break;
    // statements;
}
```

### 7.2.3 while(true)循环

while 循环测试的条件可以是任何合法的 C++ 表达式。只要条件为真，while 循环将继续执行。可以将 true 用作测试条件来创建永不结束的循环。程序清单 7.5 使用这种结构来数到 10。

程序清单 7.5 while(true)循环

```
1: // Listing 7.5
2: // Demonstrates a while true loop
3: #include <iostream>
4:
5: int main()
6: {
7:     int counter = 0;
8:
9:     while (true)
10:    {
11:        counter ++;
12:        if (counter > 10)
13:            break;
14:    }
15:    std::cout << "Counter: " << counter << std::endl;
16:    return 0;
17: }
```

**▼ 输出：**

Counter: 11

**▼ 分析：**

在第 9 行，使用一个永远不可能为假的条件来设置 while 循环。在循环体中，第 11 行对变量 counter 进行递增，然后第 12 行检测 counter 是否超过 10，如果没有，继续执行 while 循环。如果 counter 大于 10，第 13 行的 break 语句终止循环，程序跳到第 15 行继续执行：打印结果。

该程序虽然可行，但并不恰当。这是一个使用错误工具来完成工作的典范。将测试 counter 的代码放到它本应该的位置（while 条件中）可完成这样的工作。

**警告**

如果终止条件永远得不到满足，像 while(true) 这样的死循环可能导致计算机挂起。请慎用它们并进行仔细的测试。

对于同样的任务，C++ 提供了很多方法。真正的技巧在于，选择正确的工具来完成特定的工作。

应该	不应该
使用 while 循环时指定一个条件语句。 慎用 continue 和 break 语句。 确保循环最终能够结束。	不要使用 goto 语句。 别忘了 continue 和 break 之间的差别;前者跳转到循环开头,后者跳出循环。

### 7.3 实现 do...while 循环

while 循环体可能永远都不会执行。while 语句在执行循环体之前检查条件,如果条件为假,将跳过整个 while 循环体。程序清单 7.6 演示了这种情况。

程序清单 7.6 跳过 while 循环体

```
1: // Listing 7.6
2: // Demonstrates skipping the body of
3: // the while loop when the condition is false.
4:
5: #include <iostream>
6:
7: int main()
8: {
9:     int counter;
10:    std::cout << "How many hellos?: ";
11:    std::cin >> counter;
12:    while (counter > 0)
13:    {
14:        std::cout << "Hello!\n";
15:        counter--;
16:    }
17:    std::cout << "Counter is OutPut: " << counter;
18:    return 0;
19: }
```

▼ 输出:

```
How many hellos?: 2
Hello!
Hello!
Counter is OutPut: 0

How many hellos?: 0
Counter is OutPut: 0
```

▼ 分析:

第 10 行提示用户输入一个开始值。这个值被存储在 int 变量 counter 中。第 12 行对 counter 进行测试,while 循环体对 counter 进行递减。从上述输出可知,第一次运行时,counter 被设置为 2,因此 while 循环体执行两次。然而,第二次运行时,由于输入的是 0,第 12 行对 counter 进行测试时,条件不满足:counter 不大于 0,因此跳过整个 while 循环体,没有打印单词 Hello。

如果希望至少打印 Hello 一次,该如何办呢? while 循环不能满足这种要求,因为条件测试是在执行任何打印之前进行的。要确保循环体至少执行一次,可在 while 循环之前加上下面的 if 语句:

```
if (counter < 1) // force a minimum value
    counter = 1;
```

但这种做法被程序员称为“拙作 (kludge)”,是一种既难看又不雅的解决方案。

### 7.4 使用 do...while

do...while 循环在测试条件前执行循环体,从而确保循环体至少被执行一次。程序清单 7.7 使用

do...while 循环重写了程序清单 7.6 中的程序。

程序清单 7.7 do...while 循环

```

1: // Listing 7.7
2: // Demonstrates do while
3:
4: #include <iostream>
5:
6: int main()
7: {
8:     using namespace std;
9:     int counter;
10:    cout << "How many hellos? ";
11:    cin >> counter;
12:    do
13:    {
14:        cout << "Hello\n";
15:        counter--;
16:    } while (counter > 0 );
17:    cout << "Counter is: " << counter << endl;
18:    return 0;
19: }
```

#### ▼ 输出:

```

How many hellos? 2
Hello
Hello
Counter is: 0
```

#### ▼ 分析:

和前一个程序一样，程序清单 7.7 在控制台上打印单词 Hello 指定的次数，但不同的是，该程序至少会打印一次。

第 10 行提示用户输入一个值，它被保存在 int 变量 counter 中。在 do...while 循环中，在测试条件前首先执行循环体，从而确保循环体至少执行一次。第 14 行打印消息 Hello，第 15 行对 counter 进行递减。最后，第 16 行测试循环条件。如果条件为真，跳到循环开头（第 13 行）执行，否则跳到第 17 行执行。

在 do...while 循环中，continue 和 break 语句的作用与 while 循环中一样。while 循环和 do...while 循环的唯一差别在于何时测试条件。

#### do...while 语句

do...while 语句的语法如下所示：

```

do
    statement
while (condition);
```

首先执行 statement，然后计算 condition。如果 condition 为真，就执行循环；否则循环终止。其中 statement 和 condition 与 while 循环中完全相同。

示例 1:

```

// count to 10
int x = 0;
do
    cout << "X: " << x++;
while (x < 10)
```

示例 2:

```

// print lowercase alphabet.
char ch = 'a';
do
{
    cout << ch << ' ';
    ch++;
} while ( ch <= 'z' );
```

应该	不应该
<p>要确保循环体至少执行一次,请使用 do...while。</p> <p>要确保初始条件为假时不执行循环体,使用 while 循环。</p> <p>务必测试所有的循环,确保它们按期望的方式运行。</p>	<p>除非可确保代码的功能清晰易懂,否则不要在循环中使用 break 和 continue。总会有更清晰的方法来完成同样的任务。</p> <p>不要使用 goto 语句。</p>

## 7.5 for 循环

使用 while 循环进行编程时,通常需要 3 步:设置初始条件、测试条件是否为真、在每次循环中修改条件变量。程序清单 7.8 说明这一点。

程序清单 7.8 重新审视 while 循环

```
1: // Listing 7.8
2: // Looping with while
3:
4: #include <iostream>
5:
6: int main()
7: {
8:     int counter = 0;
9:
10:    while(counter < 5)
11:    {
12:        counter++;
13:        std::cout << "Looping! ";
14:    }
15:
16:    std::cout << "\nCounter: " << counter << std::endl;
17:    return 0;
18: }
```

▼ 输出:

```
Looping! Looping! Looping! Looping! Looping!
Counter: 5.
```

▼ 分析:

在该程序清单中,读者可以看到这 3 个步骤。首先,第 8 行设置了初始条件:将 counter 初始化为 0。第 10 行对条件进行测试:检测 counter 是否小于 5。最后,第 12 行对 counter 进行递增。该循环在第 13 行打印一条简单消息。读者可以想象得到,除将 counter 递增外,还可以做更重要的工作。

for 循环将这 3 个步骤(初始化、测试和递增)合并到一条语句中。for 语句由关键字 for 和一对括号组成,括号中是 3 条用分号分隔的语句:

```
for( initialization; test ; action )
{
    ...
}
```

第 1 个表达式 initialization (初始化)为初始条件,可以是任何合法的 C++语句,但通常用于创建并初始化 1 个计数变量。第 2 个表达式 test 进行测试,可以是任何合法的 C++表达式,其功能与 while 循环中的条件相同。第 3 个表达式 action 是要执行的操作,虽然可以是任何合法的 C++语句,但通常是将计数变量递增或递减。程序清单 7.9 通过重写程序清单 7.8 演示了 for 循环。



## 程序清单 7.9 for 循环

```

1: // Listing 7.9
2: // Looping with for
3:
4: #include <iostream>
5:
6: int main()
7: {
8:     int counter;
9:     for (counter = 0; counter < 5; counter++)
10:         std::cout << "Looping! ";
11:
12:     std::cout << "\nCounter: " << counter << std::endl;
13:     return 0;
14: }

```

## ▼ 输出:

```

Looping! Looping! Looping! Looping! Looping!
Counter: 5.

```

## ▼ 分析:

第9行的 for 语句将初始化 counter、测试它是否小于 5 以及对其进行递增组合到一行中。第10行为 for 语句的循环体，当然循环体也可以是一个语句块。

## for 语句

for 语句的语法如下所示:

```

for (initialization; test; action )
    statement;

```

initialization 语句用来初始化计数变量或为循环作准备。test 可以是任何 C++ 表达式，每次循环都将对其进行测试。如果 test 为真，就执行 for 循环体，然后执行 action 语句（通常是对计数器进行递增）。

示例 1:

```

// print Hello ten times
for (int i = 0; i < 10; i++)
    cout << "Hello! ";

```

示例 2:

```

for (int i = 0; i < 10; i++)
{
    cout << "Hello!" << endl;
    cout << "the value of i is: " << i << endl;
}

```

## 7.5.1 高级 for 循环

for 语句功能强大且非常灵活。3 条独立的语句（initialization、test 和 action）使其可以有多种变体。

## 1. 对多个变量进行初始化和递增

初始化多个变量、测试复合逻辑表达式、执行多条语句的情况很常见。initialization 语句和 action 语句可以用逗号分隔的多条 C++ 语句。程序清单 7.10 演示了如何对两个变量进行初始化和递增。



程序清单 7.10 initialization 和 action 可以是多条语句

```
1: //Listing 7.10
2: // Demonstrates multiple statements in
3: // for loops
4: #include <iostream>
5:
6: int main()
7: {
8:
9:     for (int i=0, j=0; i<3; i++, j++)
10:         std::cout << "i: " << i << " j: " << j << std::endl;
11:     return 0;
12: }
```

**▼ 输出:**

```
i: 0 j: 0
i: 1 j: 1
i: 2 j: 2
```

**▼ 分析:**

在第 9 行, 将两个变量 *i* 和 *j* 都初始化为 0。使用逗号将两个表达式分开, 另外, 用分号将初始化部分和条件测试部分分开。

程序运行时, 对条件 (*i*<3) 进行测试, 由于它为真, 因此执行 for 语句的循环体: 打印这两个变量的值。最后, 执行 for 语句的第 3 个子句。正如读者看到的, 这个子句也包含两个表达式, 这里是对 *i* 和 *j* 进行递增。

执行第 10 行后, 再次测试条件。如果它仍为真, 就重复执行操作 (对 *i* 和 *j* 进行递增), 然后再次执行循环体。这种过程将一直持续下去, 直到测试条件为假, 进而不再执行操作语句并结束循环。

**2. 在 for 语句中使用空语句**

for 语句中的任何一条语句都可以省略。为此, 可使用空语句, 空语句用分号表示。通过使用空语句, 可以省略 for 语句中的第 1 个和第 3 个子句, 创建一个行为与 while 循环相同的 for 循环。程序清单 7.11 演示了这一点。

程序清单 7.11 在 for 语句中使用空语句

```
1: // Listing 7.11
2: // For loops with null statements
3:
4: #include <iostream>
5:
6: int main()
7: {
8:     int counter = 0;
9:
10:    for( ; counter < 5; )
11:    {
12:        counter++;
13:        std::cout << "Looping! ";
14:    }
15:
16:    std::cout << "\nCounter: " << counter << std::endl;
17:    return 0;
18: }
```

**▼ 输出:**

```
Looping! Looping! Looping! Looping! Looping!
Counter: 5.
```

## ▼ 分析:

读者可能注意到了, 这里的 for 循环与程序 7.8 中的 while 循环非常像。第 8 行对变量 counter 进行初始化。第 10 行的 for 语句没有初始化任何值, 但包含测试 counter < 5。for 语句中没有递增子句, 因此该 for 语句的功能与下述代码相同:

```
while (counter < 5)
```

这再次表明, C++ 提供了多种完成同一项工作的方法。经验丰富的程序员不会像程序清单 7.11 那样使用 for 循环, 但它演示了 for 循环的灵活性。实际上, 通过使用 break 和 continue 语句, 完全可以创建不包含上述 3 条语句中任何一条的 for 循环。程序清单 7.12 演示了如何创建。

## 程序清单 7.12 空的 for 语句

```
1: //Listing 7.12 illustrating
2: //empty for loop statement
3:
4: #include <iostream>
5:
6: int main()
7: {
8:     int counter=0;        // initialization
9:     int max;
10:    std::cout << "How many hellos? ";
11:    std::cin >> max;
12:    for (;;)                // a for loop that doesn't end
13:    {
14:        if (counter < max)    // test
15:        {
16:            std::cout << "Hello! " << std::endl;
17:            counter++;        // increment
18:        }
19:        else
20:            break;
21:    }
22:    return 0;
23: }
```

## ▼ 输出:

```
How many hellos? 3
Hello!
Hello!
Hello!
```

## ▼ 分析:

该 for 循环被简化到了极致。在第 12 行的 for 语句中, 省略了初始化、测试和操作部分。初始化是在 for 循环之前的第 8 行完成的。测试是在第 14 行使用一条 if 语句进行的。如果测试条件为真, 将在第 17 行执行操作: 对变量 counter 进行递增; 如果测试条件为假, 在第 20 行结束循环。

虽然这个程序看起来有些荒谬, 但有时候确实需要使用 for(;;)或 while(true)循环。本章后面讨论 switch 语句时, 读者将看到一个更合理的使用这种循环的例子。

## 7.5.2 空 for 循环

由于在 for 循环头中可以做很多事情, 有时候不需要循环体做任何事情。在这种情况下, 必须在循环体中放一条空语句 (;)。分号可以与循环头位于同一行, 但这样易于被人忽略。程序清单 7.13 演示了一种在 for 循环中使用空循环体的正确方式。

程序清单 7.13 在 for 循环中使用空循环体

---

```

1: //Listing 7.13
2: //Demonstrates null statement
3: // as body of for loop
4:
5: #include <iostream>
6: int main()
7: {
8:     for (int i = 0; i<5; std::cout << "i: " << i++ << std::endl)
9:         ;
10:    return 0;
11: }
```

---

**▼ 输出:**


---

```

i: 0
i: 1
i: 2
i: 3
i: 4
```

---

**▼ 分析:**

第 8 行的 for 循环头包含 3 条语句：初始化语句创建计数变量 i，并将它初始化为 0。条件语句测试条件 i<5，操作语句打印了 i 的值，并对它进行递增。

在 for 循环体中没有什么事情可做，因此使用空语句 (;)。注意这不是一个设计良好的 for 循环：操作语句执行的操作太多。重写成下面这样将更好：

```

8:     for (int i = 0; i<5; i++)
9:         cout << "i: " << i << endl;
```

虽然功能相同，但后者更容易理解。

### 7.5.3 循环嵌套

任何循环都可以被嵌套到另一个循环体中。外层循环每执行一次，内层循环将完整执行一遍。程序清单 7.14 使用嵌套 for 循环将符号组合成阵列。

程序清单 7.14 嵌套 for 循环

---

```

1: //Listing 7.14
2: //Illustrates nested for loops
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     int rows, columns;
9:     char theChar;
10:    cout << "How many rows? ";
11:    cin >> rows;
12:    cout << "How many columns? ";
13:    cin >> columns;
14:    cout << "What character? ";
15:    cin >> theChar;
16:    for (int i = 0; i<rows; i++)
17:    {
18:        for (int j = 0; j<columns; j++)
19:            cout << theChar;
20:        cout << endl;
21:    }
22:    return 0;
23: }
```

---

资源解密网  
PDG

## ▼ 输出:

```
How many rows? 4
How many columns? 12
What character? X
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
```

## ▼ 分析:

这个程序提示用户输入行数和列数以及要打印的字符。第一个 for 循环（第 16 行）将计数变量 *i* 初始化为 0，然后执行外循环的循环体。

在第 18 行（外层 for 循环体的第一行）创建另一个循环。将另一个计数变量 *j* 初始化为 0，然后执行内层 for 循环的循环体。第 19 行打印用户指定的字符，然后返回到内层 for 循环的开头。注意内层 for 循环体只有一条语句（打印字符）。对条件 *j* < columns 进行测试，如果条件为真，就对 *j* 进行递增，并打印下一个字符。这一过程不断重复下去，直到 *j* 等于列数为止。

内层循环的条件不满足后（这里为打印 12 个 X 后），程序跳到第 20 行执行：换行，返回到外层循环头，并检测条件 *i* < rows。如果条件为真，就对 *i* 进行递增，再次执行外层循环的循环体。

在外层循环的第二次迭代中，重新执行内层循环。因此，*j* 被重新初始化为 0，再次执行整个内层循环。

这里的重要概念是，通过使用嵌套循环，外循环的每次迭代中，整个内循环都将执行一次。因此，在每一行，字符都被打印 columns 次。

## 注意

顺便说一句，很多 C++ 程序员使用 *i* 和 *j* 作为计数变量。这种传统可以追溯到 FORTRAN 语言中，在这种语句中，只能将 *i*、*j*、*k*、*l*、*m* 和 *n* 用作计数变量。

虽然这无伤大雅，但程序的读者可能对计数变量的用途感到迷惑，进而错误地使用它。您甚至会对包含嵌套循环的复杂程序感到迷惑。因此，最好通过计数变量的名称来指出其用途，例如使用 CustomerIndex 或 InputCounter。

## 7.5.4 for 循环中声明的变量的作用域

以前，在 for 循环中声明的变量的作用域为外层语句块。ANSI 标准做了修改，规定这些变量作用域为 for 循环本身的语句块，但并非所有的编译器都支持这种改变。读者可以使用以下代码测试自己的编译器：

```
#include <iostream>
int main()
{
    // i scoped to the for loop?
    for (int i = 0; i < 5; i++)
    {
        std::cout << "i: " << i << std::endl;
    }

    i = 7; // integer 'i' should not be in scope!
    return 0;
}
```

如果能够通过编译，表明您的编译器不支持 ANSI 标准所做的修改。

如果您的编译器指出 *i* 没有定义（针对 *i* = 7 所在的行），表明您的编译器支持新标准。如果像如下所示在循环外声明变量 *i*，则无论编译器是否支持新标准，代码都能够通过编译：

```

#include <iostream>
int main()
{
    int i; //declare outside the for loop
    for (i = 0; i<5; i++)
    {
        std::cout << "i: " << i << std::endl;
    }

    i = 7; // now this is in scope for all compilers
    return 0;
}

```

## 7.6 循环小结

第 6 章介绍了如何使用递归解决 Fibonacci 数列问题。这里简要地回顾一下，Fibonacci 数列以 1、1、2、3 开始，所有后续数字都是前两个数字的和：

1, 1, 2, 3, 5, 8, 13, 21, 34...

第  $n$  个 Fibonacci 数是第  $n-1$  个和第  $n-2$  个 Fibonacci 数之和。第 6 章解决的问题是计算第  $n$  个 Fibonacci 数，这是通过递归实现的。程序清单 7.15 提供了一个使用循环的解决方案。

程序清单 7.15 使用循环计算第  $n$  个 Fibonacci 数

---

```

1: // Listing 7.15 - Demonstrates solving the nth
2: // Fibonacci number using iteration
3:
4: #include <iostream>
5:
6: unsigned int fib(unsigned int position );
7: int main()
8: {
9:     using namespace std;
10:    unsigned int answer, position;
11:    cout << "Which position? ";
12:    cin >> position;
13:    cout << endl;
14:
15:    answer = fib(position);
16:    cout << answer << " is the ";
17:    cout << position << "th Fibonacci number. " << endl;
18:    return 0;
19: }
20:
21: unsigned int fib(unsigned int n)
22: {
23:     unsigned int minusTwo=1, minusOne=1, answer=2;
24:
25:     if (n < 3)
26:         return 1;
27:
28:     for (n -= 3; n != 0; n--)
29:     {
30:         minusTwo = minusOne;
31:         minusOne = answer;
32:         answer = minusOne + minusTwo;
33:     }
34:
35:     return answer;
36: }

```

---

### ▼ 输出：

```

Which position? 4
3 is the 4th Fibonacci number.
Which position? 5
5 is the 5th Fibonacci number.

```





其中 expression 可以是任何合法的 C++ 表达式, statement 可以是任何合法的 C++ 语句或语句块。注意, case 值必须为整数或结果为整数的表达式,但在这种表达式中不能使用关系运算符和布尔运算符。

如果某个 case 值与表达式匹配,程序跳转到该 case 后面的语句处执行,直到遇到 break 语句或到达 switch 语句块末尾为止。如果没有匹配的值,程序跳转到默认 (default) 分支处执行。如果既没有匹配的值也没有默认分支,程序将跳出 switch 语句。

**注意**

在 switch 语句中提供一个 default 分支总是个好主意。如果本不需要 default 分支,可使用它来检测不可能出现的情况,并打印错误消息。这对调试很有帮助。

需要注意的时,如果 case 语句后面没有 break 语句,程序将继续执行下一条 case 语句。虽然有时候需要这样做,但通常是错误。如果需要对程序继续向下执行,请用注释来指出您并非忘记了使用 break 语句。

程序清单 7.16 演示了 switch 语句的用法。

程序清单 7.16 switch 语句的用法

```
1: //Listing 7.16
2: // Demonstrates switch statement
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     unsigned short int number;
9:     cout << "Enter a number between 1 and 5: ";
10:    cin >> number;
11:    switch (number)
12:    {
13:        case 0:    cout << "Too small, sorry!";
14:                  break;
15:        case 5:    cout << "Good job! " << endl; // fall through
16:        case 4:    cout << "Nice Pick!" << endl; // fall through
17:        case 3:    cout << "Excellent!" << endl; // fall through
18:        case 2:    cout << "Masterful!" << endl; // fall through
19:        case 1:    cout << "Incredible!" << endl;
20:                  break;
21:        default:   cout << "Too large!" << endl;
22:                  break;
23:    }
24:    cout << endl << endl;
25:    return 0;
26: }
```

**▼ 输出:**

```
Enter a number between 1 and 5: 3
Excellent!
Masterful!
Incredible!

Enter a number between 1 and 5: 8
Too large!
```

**▼ 分析:**

第 9 行和第 10 行提示用户输入一个数字,第 11 行将这个数字用于 switch 语句中。如果数字为 0,将与第 13 行的 case 语句匹配,进而打印消息 Too small, sorry!,第 14 行的 break 语句跳出 switch 语句。如果数字为 5,将执行第 15 行:打印一条消息,然后执行第 16 行:打印另一条消息,依次类推,直到到达第 20 行的 break 语句:跳出 switch 语句。

这些语句的效果是,当输入值在 1 和 5 之间时,将打印多条消息。如果输入的数值不在 0 和 5 之间,则认为它太大,并执行第 21 行的默认语句。

### switch 语句

switch 语句的语法如下:

```
switch (expression)
{
    case valueOne: statement;
    case valueTwo: statement;
    ....
    case valueN: statement;
    default: statement;
}
```

switch 语句让您能够为多个值提供不同的分支。它计算表达式的值, 如果与某个 case 值匹配, 就跳到该行执行。然而继续执行, 直到到达 switch 语句的结尾或遇到 break 语句为止。

如果表达式和任何 case 值都不匹配, 且指定了默认语句, 则执行默认语句; 否则, 结束 switch 语句。

示例 1:

```
switch (choice)
{
    case 0:
        cout << "Zero!" << endl;
        break;
    case 1:
        cout << "One!" << endl;
        break;
    case 2:
        cout << "Two!" << endl;
    default:
        cout << "Default!" << endl;
}
```

示例 2:

```
switch (choice)
{
    case 0:
    case 1:
    case 2:
        cout << "Less than 3!";
        break;
    case 3:
        cout << "Equals 3!";
        break;
    default:
        cout << "greater than 3!";
}
```

## 使用 switch 语句来处理菜单

程序清单 7.17 使用了前面讨论的 for(;;) 循环。这种循环也被称为死循环, 因为如果没有到 break 语句, 将永远循环下去。在程序清单 17.17 中, 死循环被用来提供菜单, 要求用户进行选择, 然后根据用户的选择执行相应的操作并返回到菜单。循环将不断执行, 直到用户选择退出。

### 注意

有些程序员喜欢这样编写死循环:

```
#define EVER ;;
for (EVER)
{
    // statements...
}
```

死循环是没有退出条件的循环。要退出循环, 必须使用 break 语句。死循环也被称为无穷循环。

## 程序清单 7.17 死循环

```
1: //Listing 7.17
2: //Using a forever loop to manage user interaction
3: #include <iostream>
4:
5: // prototypes
6: int menu();
7: void DoTaskOne();
8: void DoTaskMany(int);
9:
10: using namespace std;
11:
12: int main()
13: {
14:     bool exit = false;
15:     for (;;)
16:     {
17:         int choice = menu();
18:         switch(choice)
19:         {
20:             case (1):
21:                 DoTaskOne();
22:                 break;
23:             case (2):
24:                 DoTaskMany(2);
25:                 break;
26:             case (3):
27:                 DoTaskMany(3);
28:                 break;
29:             case (4):
30:                 continue; // redundant!
31:                 break;
32:             case (5):
33:                 exit=true;
34:                 break;
35:             default:
36:                 cout << "Please select again! " << endl;
37:                 break;
38:         } // end switch
39:
40:         if (exit == true)
41:             break;
42:     } // end forever
43:     return 0;
44: } // end main()
45:
46: int menu()
47: {
48:     int choice;
49:
50:     cout << " **** Menu **** " << endl << endl;
51:     cout << "(1) Choice one. " << endl;
52:     cout << "(2) Choice two. " << endl;
53:     cout << "(3) Choice three. " << endl;
54:     cout << "(4) Redisplay menu. " << endl;
55:     cout << "(5) Quit. " << endl << endl;
56:     cout << ": ";
57:     cin >> choice;
58:     return choice;
59: }
60:
61: void DoTaskOne()
62: {
63:     cout << "Task One! " << endl;
64: }
65:
66: void DoTaskMany(int which)
67: {
68:     if (which == 2)
69:         cout << "Task Two! " << endl;
70:     else
71:         cout << "Task Three! " << endl;
72: }
```



▼ 输出:

```
**** Menu ****

(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 1
Task One!
**** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 3
Task Three!
**** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 5
```

▼ 分析:

该程序使用本章和前几章介绍的很多概念，还演示了 switch 语句的常见用法。

死循环从第 15 行开始。该循环调用 menu( )函数，向屏幕打印菜单并返回用户的选择。switch 语句从第 18 行开始，到第 38 行结束，它根据用户的选择执行相应的操作。

如果用户输入 1，程序跳到第 20 行的 case(1):语句处执行。第 21 行调用函数 DoTaskOne( )，该函数打印一条消息，然后返回。返回后，程序继续执行第 22 行：用 break 语句结束 switch 语句，程序跳到第 39 行执行。第 40 行检查变量 exit 的值是否为 true。如果是，就执行第 41 行的 break 语句，从而退出 for(;;)循环；否则，回到循环开头（第 15 行）继续执行。

注意，第 30 行中的 continue 语句是多余的。如果删除这条语句，将遇到 break 语句，从而结束 switch 语句，且 exit 的值为假，因此重新执行循环，进而打印菜单。然而，continue 确实避免了对 exit 进行测试。

应该	不应该
没有 case 语句中使用 break 时，一定要对其原因进行说明。	如果更清晰的 switch 语句可行，不要使用复杂的 if...else 语句。
在 switch 语句中一定要包含 default 语句，哪怕只是为了检查看似不可能的情形。	别忘了在每个 case 的末尾加上 break 语句，除非您希望继续向下执行。

7.8 总结

本章首先介绍了应避免使用的 goto 命令，然后介绍了各种不需要使用 goto 语句就能让 C++ 程序进行循环的方法。

while 循环首先检查条件，如果为真，则执行循环体中的语句。do...while 循环首先执行循环体，然后对条件进行测试。for 循环初始化一个值，然后测试表达式，如果为真，则执行循环体；然后，执行 for 循环头中的最后一条语句，并再次检查条件。这种检查条件、执行循环体中的语句并执行 for 循环



体中最后一条语句的过程不断持续下去，直到条件表达式为假为止。

本章还介绍了 `continue` 语句和 `break` 语句，前者导致 `while`、`do...while` 和 `for` 循环重新开始执行，而 `break` 语句导致 `while`、`do...while`、`for` 和 `switch` 语句结束。

## 7.9 问与答

问：如何在 `if...else` 和 `switch` 语句之间做出选择？

答：如果有两个以上的 `else` 子句，且所有子句都测试相同的变量，应考虑使用 `switch` 语句。

问：如何在 `while` 和 `do...while` 语句之间进行选择？

答：如果循环体至少应执行一次，考虑使用 `do...while` 语句；否则，使用 `while` 语句。

问：如何在 `while` 和 `for` 语句之间进行选择？

答：如果要初始化计数变量，且每次循环都需要对该变量进行测试和递增，应考虑使用 `for` 循环。如果变量已经初始化，且并非每次循环都要对其进行递增，则 `while` 循环可能是更好的选择。经验丰富的程序员期望您遵循上述准则，如果您不这样做，程序将难以理解。

问：`while(true)`和`for(;;)`哪个更好？

答：这两者之间没有明显的区别，但最好避免同时使用它们。

问：为何不应像 `while(n)` 中那样将变量用作条件？

答：在最新的 C++ 标准中，表达式的结果为布尔值 `true` 或 `false`。虽然可以将 `false` 视为 0，`true` 视为其他任何值，但最好使用结果为布尔值 `true` 或 `false` 的表达式，这与最新的标准也更为一致。然而，`bool` 型变量可用作条件，而不会有任何潜在的问题。

## 7.10 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 7.10.1 测验

1. 如何在 `for` 循环中初始化多个变量？
2. 为什么要避免使用 `goto` 语句？
3. 能否编写一个循环体永远不会被执行的 `for` 循环？
4. 下面的 `for` 循环执行完毕后，`x` 的值是多少？  

```
for (int x = 0; x < 100; x++)
```
5. 在 `for` 循环中能够嵌套 `while` 循环吗？
6. 能否创建一个永不结束的循环？请举例说明。
7. 如果创建了一个永不结束的循环将发生什么情况？

### 7.10.2 练习

1. 编写一个嵌套 `for` 循环，打印由 10×10 个 0 组成的图案。

2. 编写一个 for 循环, 从 100 数到 200, 每次增加 2。
3. 编写一个 while 循环, 从 100 数到 200, 每次增加 2。
4. 写一个 do...while 循环, 从 100 数到 200, 每次增加 2。

5. 查错: 下面的代码有何错误?

```
int counter = 0;
while (counter < 10)
{
    cout << "counter: " << counter;
}
```

6. 查错: 下面的代码有何错误?

```
for (int counter = 0; counter < 10; counter++);
    cout << counter << " ";
```

7. 查错: 下面的代码有何错误?

```
int counter = 100;
while (counter < 10)
{
    cout << "counter now: " << counter;
    counter--;
}
```

8. 查错: 下面的代码有何错误?

```
cout << "Enter a number between 0 and 5: ";
cin >> theNumber;
switch (theNumber)
{
    case 0:
        doZero();
    case 1:           // fall through
    case 2:           // fall through
    case 3:           // fall through
    case 4:           // fall through
    case 5:
        doOneToFive();
        break;
    default:
        doDefault();
        break;
}
```

# 第 8 章

## 阐述指针

对于 C++ 程序员来说，一个强大而低级的工具是，可以使用指针来直接操纵计算机内存。这是 C++ 相对于诸如 Java、C# 和 Visual Basic 等其他语言的优点之一。

学习 C++ 时，指针带来了两方面的挑战：它们令人迷惑，为何需要它们的原因不那么明显。本章将循序渐进地介绍指针的工作原理。然而，只有通过继续阅读本书，读者才能完全地理解为什么需要使用指针。

在本章中，您将学习：

- 什么是指针
- 如何声明和使用指针
- 什么是自由存储（free store）以及如何操纵内存

### 8.1 什么是指针

指针是存储内存地址的变量，就这么简单。如果读者明白了这句话，就理解了有关指针的核心知识。

#### 8.1.1 内存简介

要理解指针，必须对计算机内存有一定的了解。计算机内存被划分成按顺序编号的内存单元。每个变量都位于某个独特的内存单元中，这被称为地址。图 8.1 说明了名为 theAge 的 unsigned long 变量在内存中的存储情况。

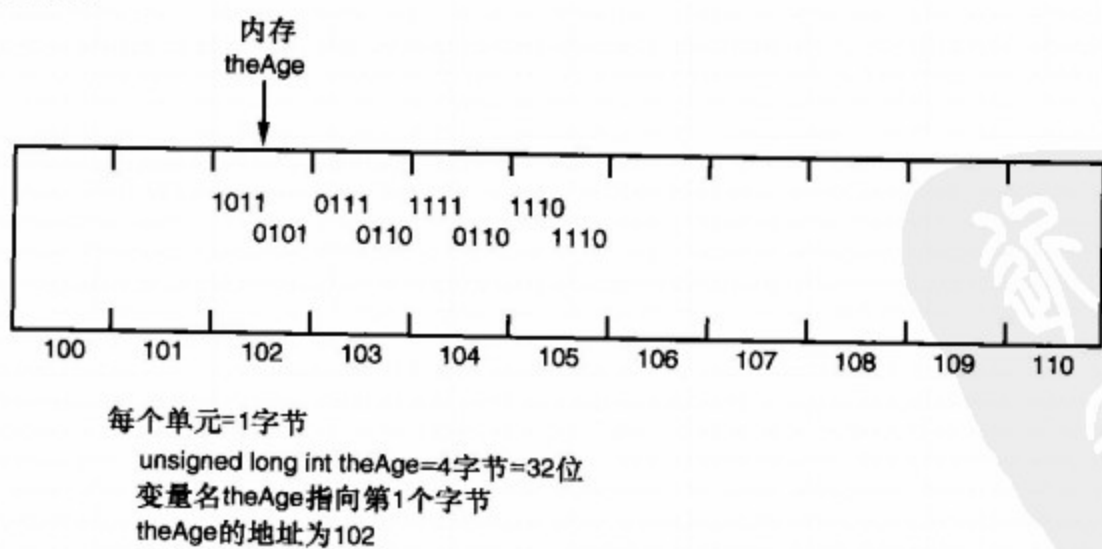


图 8.1 theAge 的存储情况

注意

能够使用指针以及在低层操纵内存是 C++ 被选择用于编写嵌入式和实时应用程序的原因之一。

### 8.1.2 获取变量的内存地址

不同的计算机使用不同的复杂方案对内存进行编号。通常，程序员不需要知道变量的具体地址，因为编译器负责处理细节。然而，要获得变量的内存地址，可以使用地址运算符`&`，它返回对象在内存中的地址。程序清单 8.1 演示了这种运算符的用法。

程序清单 8.1 地址运算符

```
1: // Listing 8.1 Demonstrates address-of operator
2: // and addresses of local variables
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     unsigned short shortVar=5;
9:     unsigned long longVar=65535;
10:    long sVar = -65535;
11:
12:    cout << "shortVar:\t" << shortVar;
13:    cout << "\tAddress of shortVar:\t";
14:    cout << &shortVar << endl;
15:
16:    cout << "longVar:\t" << longVar;
17:    cout << "\tAddress of longVar:\t" ;
18:    cout << &longVar << endl;
19:
20:    cout << "sVar:\t\t" << sVar;
21:    cout << "\tAddress of sVar:\t" ;
22:    cout << &sVar << endl;
23:
24:    return 0;
25: }
```

▼ 输出:

shortVar:	5	Address of shortVar:	0012FF7C
longVar:	65535	Address of longVar:	0012FF78
sVar:	-65535	Address of sVar:	0012FF74

读者运行该程序时，输出可能与此不同，尤其是最后一列。

▼ 分析:

声明并初始化了 3 个变量：第 8 行的 `unsigned short` 变量，第 9 行的 `unsigned long` 变量，第 10 行的 `long` 变量。第 12~22 行打印它们的值和地址通。在第 14、18 和 22 行，使用地址运算符`&`来获得变量的地址。只需将该运算符放在变量名的前面，就可以返回变量的地址。

第 12 行打印 `shortVar` 的值，结果为 5，与预期的相同。从输出中的第 1 行可知，该变量的地址为 0012FF7C，这是在一台奔腾（32 位）计算机上运行该程序得到的。该地址随计算机而异，且每次运行该程序时都可能有所不同。读者的结果将与此不同。

当您声明变量时，编译器将根据其类型决定为它分配多少内存。编译器负责为变量分配内存，并自动给它们指定地址。例如，`long` 变量通常占用 4 字节，因此使用一个指向 4 字节内存的地址。

注意

您的编译器可能总是给新变量分配 4 字节整数倍的内存，因此 `longVar` 的地址可能比 `shortVar` 的地址大 4，虽然 `shortVar` 只需要 2 字节。

### 8.1.3 将变量的地址存储到指针中

每个变量都有地址。即使不知道变量的具体地址，也可以将其地址存储到指针中。

例如, 假设 howOld 是一个整型变量。要声明一个名为 pAge 的指针来存储它的地址, 可以这样做:

```
int *pAge = 0;
```

该语句将 pAge 声明为一个 int 指针。也就是说, pAge 被声明为用于存储整型变量的地址。

注意, pAge 是一个变量。当您声明 (类型为 int 的) 整型变量时, 编译器将分配足以存储一个整数的内存。当您声明诸如 pAge 等指针变量时, 编译器将分配足以存储一个地址的内存 (在大多数计算机上为 4 字节)。指针 (如 pAge) 只不过是另一种类型的变量而已。

### 8.1.4 指针名

由于指针也是变量, 因此可以使用任何合法的变量名, 有关变量的命名规则和建议也适用于指针名。很多程序员采用这样的命名规则: 所有指针名都以 p 打头, 如 pAge 和 pNumber。

在下面的例子中:

```
int *pAge = 0;
```

pAge 被初始化为 0。值为 0 的指针被称为空指针。所有指针在创建时都应该进行初始化。如果不知道要将什么值赋给指针, 将 0 赋给它。没有被初始化的指针被称为失控指针 (wild pointer), 因为您不知道它指向的什么——可能指向任何东西! 失控指针是非常危险的。

#### 注意

一定要初始化所有指针。

要让指针存储一个地址, 必须将该地址赋给它。在前面的范例中, 必须将 howOld 的地址赋给 pAge, 如下所示:

```
unsigned short int howOld = 50;    // make a variable
unsigned short int * pAge = 0;    // make a pointer
pAge = &howOld;                  // put howOld's address in pAge
```

第 1 行创建了一个名为 howOld 的变量 (其类型为 unsigned short int), 并将其初始化为 50, 第 2 行将 pAge 声明为一个 unsigned short int 指针, 并将其初始化为 0。由于变量类型后和变量名之间有一个星号 (\*), 您知道 pAge 是一个指针。

第 3 行将 howOld 的地址赋给指针 pAge。您之所以知道赋给 pAge 的是 howOld 的地址, 是因为其中使用了地址运算符 &。如果不使用地址运算符, 赋给的将是 howOld 的值。这个值可能是有效的地址, 也可能是无效地址。

现在, pAge 的值为 howOld 的地址, 而 howOld 的值为 50。可以采取更少的步骤来完成上述工作, 如下所示:

```
unsigned short int howOld = 50;    // make a variable
unsigned short int * pAge = &howOld; // make pointer to howOld
```

### 8.1.5 获取指针指向的变量的值

使用 pAge 可获取 howOld 的值 (这里为 50)。通过指针 pAge 访问变量的值称为间接访问, 因为这是通过指针间接地访问变量。例如, 可以通过指针 pAge 间接地访问 howOld 的值。

间接访问意味着访问位于指针存储的地址处的值。指针提供了一种间接方式来获取存储在地址处的值。

#### 注意

常规变量的类型告诉编译器需要多少内存来存储其值, 而指针的类型没有这样的功能, 所有指针都存储内存地址, 因此长度都相同: 在使用 32 位处理器的计算机上为 4 字节, 在使用 64 位处理器的计算机上为 8 字节。

指针的类型告诉编译器, 它存储的地址处的对象需要多少内存。

下面的声明将 pAge 声明为一个 unsigned short int 指针:

```
unsigned short int * pAge = 0;    // make a pointer
```



这告诉编辑器，该指针（为存储地址需要4字节）存储一个类型为 unsigned short int 的对象的地址，该对象本身需要2字节。

### 8.1.6 使用间接运算符解除引用

间接运算符（\*）也被称为解除引用（dereference）运算符。对指针解除引用时，将得到指针存储的地址处的值。

常规变量让您能够直接访问它的值。要创建一个名为 yourAge 的 unsigned short int 变量，并将 howOld 的值赋给它，可以这样做：

```
unsigned short int yourAge;  
yourAge = howOld;
```

指针让您能够间接地访问其指向的变量的值。要通过指针将 howOld 的值赋给变量 yourAge，可以这样做：

```
unsigned short int yourAge;  
yourAge = *pAge;
```

指针变量 pAge 前面的间接运算符（\*）的含义是：存储在该地址处的值。这条赋值语句的含义是：将存储在 pAge 中的地址处的值赋给 yourAge。如果像下面这样不使用间接运算符：

```
yourAge = pAge; // bad!!
```

则是试图将 pAge 的值（一个内存地址）赋给 yourAge。编译器很可能会对此提出警告：指出您可能在犯错。

#### 星号的其他用途

用于指针时，星号的用途有两种：声明指针和用作解除引用运算符。

声明指针时，\*是声明的组成部分，位于被指向的对象的类型后面，例如：

```
// make a pointer to an unsigned short  
unsigned short * pAge = 0;
```

对指针解除引用时，解除引用（间接）运算符指出要访问指针指向的内存单元中的值，而不是地址本身。

```
// assign 5 to the value at pAge  
*pAge = 5;
```

另外，星号（\*）还可用作乘法运算符。编译器能够根据您使用星号（上下文）知道该调用哪个运算符。

### 8.1.7 指针、地址和变量

对指针、其存储的地址以及其存储的地址处的值进行区分至关重要。这是人们对指针感到迷惑的主要根源。

请看下面的代码段：

```
int theVariable = 5;  
int * pPointer = &theVariable ;
```

theVariable 被声明为一个 int 变量，并被初始化为 5；pPointer 被声明为一个 int 指针，并被初始化为 theVariable 的地址。pPointer 是一个指针，存储的是 theVariable 的地址。pPointer 存储的地址处的值为 5。图 8.2 说明了 theVariable 和 pPointer。

在图 8.2 中，5 存储在地址为 101 的内存单元中。5 的二进制表示如下：

```
0000 0000 0000 0101
```

这个二进制数长两字节（16 位），对应的十进制数为 5。

指针变量存储在内存单元 106 中，其值如下：

```
000 0000 0000 0000 0000 0000 0110 0101
```

这是十进制数 101 的二进制表示，它是变量 theVariable 的地址，该变量的值为 5。

这里的内存布局是示意性的，但它说明了指针是如何存储地址的。

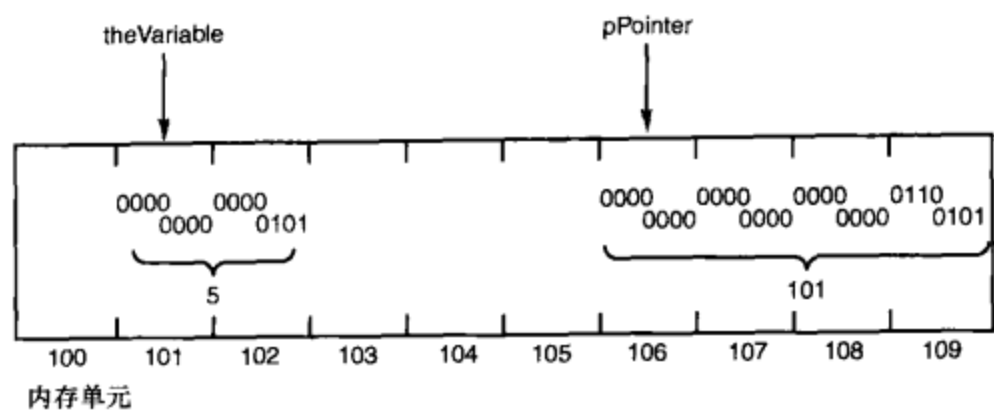


图 8.2 theVariable 和 pPointer 在内存中的情况

8.1.8 使用指针来操纵数据

除使用间接运算符来查看存储在指针指向的位置处的数据外，还可以对数据进行操纵。将变量的地址赋给指针后，可以使用该指针来访问它指向的变量中的数据。

程序清单 8.2 使用了刚才介绍的全部有关指针的指针，它演示了如何将一个局部变量的地址赋给指针以及如何使用指针和间接运算符来操纵该变量的值。

程序清单 8.2 使用指针来操纵数据

```
1: // Listing 8.2 Using pointers
2: #include <iostream>
3:
4: typedef unsigned short int USHORT;
5:
6: int main()
7: {
8:
9:     using namespace std;
10:
11:     USHORT myAge;           // a variable
12:     USHORT * pAge = 0;     // a pointer
13:
14:     myAge = 5;
15:
16:     cout << "myAge: " << myAge << endl;
17:     pAge = &myAge;         // assign address of myAge to pAge
18:     cout << "*pAge: " << *pAge << endl << endl;
19:
20:     cout << "Setting *pAge = 7... " << endl;
21:     *pAge = 7;              // sets myAge to 7
22:
23:     cout << "*pAge: " << *pAge << endl;
24:     cout << "myAge: " << myAge << endl << endl;
25:
26:     cout << "Setting myAge = 9... " << endl;
27:     myAge = 9;
28:
29:     cout << "myAge: " << myAge << endl;
30:     cout << "*pAge: " << *pAge << endl;
31:
32:     return 0;
33: }
```

▼ 输出:

myAge: 5
\*pAge: 5

Setting \*pAge = 7...
\*pAge: 7
myAge: 7

Setting myAge = 9...
myAge: 9
\*pAge: 9

▼ 分析:

该程序声明了两个变量：unsigned short 变量 myAge 和 unsigned short 指针 pAge。第 14 行将 5 赋给 myAge，第 16 行的打印输出验证了这一点。

第 17 行将 myAge 的地址赋给 pAge。第 18 行使用间接运算符 (\*) 对 pAge 解除引用并打印结果, 结果表明, pAge 存储的地址处的值是存储在 myAge 中的 5。

第 21 行将 7 赋给 pAge 指向的变量，这相当于将 myAge 的值设置为 7，第 23 行和第 24 行证明了这一点。间接访问变量是使用星号（间接运算符）实现的。

第 27 行将 9 赋给变量 myAge。第 29 行直接获取这个值，而第 30 行间接（通过对 pAge 解除引用）获得这个值。

### 8.1.9 查看地址

指针让您能够操纵地址，而无需知道其实际值。将变量的地址赋给指针时，指针的值将为变量的地址，阅读本章后，读者将对此深信不疑。程序清单 8.3 说明了这一点。

### 程序清单 8.3 确定指针中存储的内容

```

1: // Listing 8.3
2: // What is stored in a pointer.
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:
9:     unsigned short int myAge = 5, yourAge = 10;
10:
11:     // a pointer
12:     unsigned short int * pAge = &myAge;
13:
14:     cout << "myAge:\t" << myAge
15:         << "\t\tyourAge:\t" << yourAge << endl;
16:
17:     cout << "&myAge:\t" << &myAge
18:         << "\t&yourAge:\t" << &yourAge << endl;
19:
20:     cout << "pAge:\t" << pAge << endl;
21:     cout << "*pAge:\t" << *pAge << endl;
22:
23:
24:     cout << "\nReassigning: pAge = &yourAge..." << endl << endl;
25:     pAge = &yourAge;           // reassign the pointer
26:
27:     cout << "myAge:\t" << myAge <<
28:         "\t\tyourAge:\t" << yourAge << endl;
29:
30:     cout << "&myAge:\t" << &myAge
31:         << "\t&yourAge:\t" << &yourAge << endl;
32:
33:     cout << "pAge:\t" << pAge << endl;
34:     cout << "*pAge:\t" << *pAge << endl;
35:
36:     cout << "\n&pAge:\t" << &pAge << endl;
37:
38:     return 0;
39: }

```

▼ 输出:

```
myAge: 5          yourAge: 10
&myAge: 0012FF7C  &yourAge: 0012FF78
pAge: 0012FF7C
```

```
*pAge: 5

Reassigning: pAge = &yourAge...

myAge: 5          yourAge: 10
&myAge: 0012FF7C  &yourAge: 0012FF78
pAge: 0012FF78
*pAge: 10

&pAge: 0012FF74
```

读者的输出可能与此不同。

▼ 分析:

第 9 行将 myAge 和 yourAge 声明为 unsigned short int 变量，第 12 行将 pAge 声明为 unsigned short int 指针，并将其初始化为变量 myAge 的地址。

第 14~18 行打印了 myAge、yourAge 的值和地址。第 20 行打印 pAge 的内容：myAge 的地址。第 21 行打印对 pAge 解除引用的结果，即打印 pAge 指向的变量 myAge 的值 5。

这就是指针的实质。第 20 行表明，pAge 存储的是 myAge 的地址；第 21 行演示了如何通过对指针 pAge 解除引用来获得 myAge 的值。继续学习后面的内容之前，一定要完全了解这些概念。请仔细研究代码并查看输出。

第 25 行重新给 pAge 赋值，使之指向 yourAge。然后，再次打印值和地址。输出表明，pAge 现在存储的是变量 yourAge 的地址，而对 pAge 解除引用得到的是 yourAge 的值。

第 36 行打印 pAge 本身的地址。像其他任何变量一样，它也有地址，在该地址处可以存储一个指针。稍后将讨论将指针的地址赋给另一个指针。

应该	不应该
务必使用间接运算符 (*) 来访问存储在指针指向的地址处的值。 务必将任何指针初始化为合法的地址或空值 (0)。	不要将指针指向的地址与存储在该地址处的值混为一谈。

8.1.10 指针和数组名

在 C++ 中，数组名是一个常量指针，指向数组的第一个元素。因此，在下列声明中：

```
int Numbers [5];
```

Number 是一个指向数组 Number 的第一个元素 Family[0] 的指针。

将数组名用作常量指针是合法的，反之亦然。因此，Number + 4 是一种访问 Number [4] 的合法方式。

当您 对指针执行加法、递增、递减运算时，编译器将执行所有的算术运算。Number + 4 指向的是 Number 中的第 4 个对象（即第 4 个整数），而不是离 Number 开头 4 字节处。由于每个整数通常占 4 字节，因此 Number + 4 指向离数组开头 16 字节处。

程序清单 8.4 使用指针来打印数组的内容，以演示指针和数组之间的关系。

程序清单 8.4 数组和指针之间的关系

```
0: #include <iostream>
1: const int ARRAY_LENGTH = 5;
2:
3: int main ()
4: {
```

```
5: using namespace std;
6:
7: // An array of 5 integers initialized to 5 values
8: int Numbers [ARRAY_LENGTH] = {0, 100, 200, 300, 400};
9:
10: // pInt points to the first element
11: const int *pInt = Numbers;
12:
13: cout << "Using a pointer to print the contents of the array: " << endl;
14:
15: for (int nIndex = 0; nIndex < ARRAY_LENGTH; ++ nIndex)
16:     cout << "Element [" << nIndex << "] = " << *(pInt + nIndex) << endl;
17:
18: return 0;
19: }
```

▼ 输出:

```
Using a pointer to print the contents of the array:
Element [0] = 0
Element [1] = 100
Element [2] = 200
Element [3] = 300
Element [4] = 400
```

▼ 分析:

该程序创建了一个名为 Numbers 的数组（它包含 5 个整数），然后使用指针 pInt 访问该数组的元素。数组名（这里为 Numbers）是指向第一个元素的指针，pInt 也如此。因此(pInt + 1)指向第二个元素，(pInt + 2)指向第三个元素，依此类推。通过对这些指针解除引用，可获取指向的值。因此，\*pInt 返回第一个元素的值，\*(pInt + 1)返回第二个元素的值，依此类推。

这个示例说明了指针和数组之间的相似性。事实上，在第 16 行，可使用 pInt [nIndex]代替\*(pInt + nIndex)，这将得到相同的输出。

8.1.11 数组指针和指针数组

请看下面 3 个声明：

```
int NumbersOne[500];
int * NumbersTwo[500];
int * NumbersThree = new int[500];
```

NumbersOne 是一个包含 500 个 int 对象的数组，NumbersTwo 是一个包含 500 个 int 指针的数组，NumbersThree 是一个指针，指向一个包含 500 个 int 对象的数组。

前面说过，数组名（如 NumbersOne）实际上是一个指针，它指向第一个数组元素。因此，第 1 个和第 3 个声明（即 NumbersOne 和 NumbersThree）最为相似，而 NumbersTwo 是一个指针数组，该数组包含指向整数的指针，即它包含 500 个地址，这些地址可指向内存中的整数。

应该	不应该
使用间接运算符 (*) 来访问存储在指针指向的地址处的数据。 将所有指针初始化为有效的地址或 null (0)。	不要将指针指向的地址与存储在地址处的值混为一谈。

使用指针

要声明指针，首先指出指针将指向的变量或对象的类型，然后加上指针运算符 (\*) 和指针名。例如：



```
unsigned short int * pPointer = 0;
```

要给指针赋值或初始化，在要将其地址赋给指针的变量名前加上地址运算符&。例如：

```
unsigned short int theVariable = 5;  
unsigned short int * pPointer = &theVariable;
```

要对指针解除引用，在指针名前使用解除引用运算符(\*)。例如：

```
unsigned short int theValue = *pPointer
```

## 8.2 为什么使用指针

至此，读者已经知道了将变量的地址赋给指针的步骤，然而，在实际编程中，您不会这样做。既然使用变量就可以访问数据，为什么还要使用指针呢？使用指针来操纵自动变量的唯一原因是，为了说明指针的工作原理。现在，读者已经熟悉了指针的语法，可以将其用于更好的用途。指针最常被用于完成下列 3 种任务：

- 管理自由存储区中的数据；
- 访问类的成员数据和函数；
- 按引用传递参数。

本章余下的内容将重点介绍管理自由存储区中的数据和访问类的成员数据和函数。下一章将介绍使用指针来传递变量，这被称为按引用传递。

## 8.3 栈和自由存储区（堆）

在 6.13 节中，提到了 5 个内存区域：

- 全局名称空间；
- 自由存储区；
- 寄存器；
- 代码空间；
- 堆栈。

局部变量和函数参数位于堆栈中；当然，代码位于代码空间中；而全局变量位于全局名称空间中；寄存器用于内部管理工作，如记录栈顶指针和指令指针。余下的所有内存都被作为自由存储区，通常被称为堆。

局部变量不是永久性的，函数返回时，局部变量就被删除。这很好，因为这意味着根本不用为管理这种内存空间而劳神；也不好，因为这使得函数在不将堆中的对象复制到调用函数中的目标对象的情况下，将难以创建供其他对象或函数使用的对象。全局变量解决了这种问题，其代价是在整个程序中都可以访问它们，这导致创建的程序难以理解和维护。如果管理得当，将数据存储的自由存储区可以解决这两种问题。

可以将自由存储区视为一块很大的内存，其中有数以千计的依次被编号的内存单元，可用于存储数据。与堆栈不同，您不能对这些单元进行标记，而必须先申请内存单元的地址，然后将它存储到指针中。

可以使用这样的类比：朋友给了您 Acme Mail Order 的 800 电话号码。您回到家中，将该电话号码与某个按钮绑定，然后扔掉记录电话号码的纸张。如果按下这个按钮，被拨打的电话将响铃，Acme Mail Order 公司的职员进行接听。您不需要记下这个电话号码，也不知道被拨打的电话在哪里，但只要按下绑定到的按钮就能致电 Acme Mail Order 公司。自由存储区中的数据就像 Acme Mail Order 公司一样。您不知道它在什么地方，但知道如何找到它。您使用地址（在这个例子中，地址为电话号码）访问它。您不必知道地址，只需将其放在一个指针（按钮）中。指针让您能够访问数据，而不必知道细节。

函数返回时，堆栈被自动清空。所有局部变量都不在作用域内，它们被从堆栈中删除。程序结束

前，自由存储区不会被清空，程序员使用完自己分配的内存后，必须负责将其释放。

自由存储区的优点是，您从中分配的内存将一直可用，直到您明确地指出不再需要——将其释放。如果在函数中分配自由存储区中的内存，在函数返回后该内存仍可用。

这也是自由存储区的缺点，如果您忘记释放内存，被占据而没有使用的内存将随着时间的推移越来越多，导致系统崩溃。

采取这种内存访问方式而不是全局变量的优点是，只有能够访问指针的函数才能访问它指向的数据。这样，只有将包含指针的对象或指针本身传递给函数，函数才能修改指针指向的数据，从而减少了函数能够修改数据而又无法跟踪变更的情况发生。

要做到这一点，必须能够创建指向自由存储区中某个区域的指针，并能够在函数之间传递该指针。接下来的几节介绍这是如何实现的。

### 8.3.1 使用关键字 new 分配内存

在 C++ 中，使用关键字 new 来分配自由存储区中的内存。在 new 后面跟上要为其分配内存的对象的类型，让编译器知道需要多少内存。因此，new unsigned short int 在自由存储区中分配两字节内存，而 new unsigned long 分配 4 字节内存。这里假设在您的系统中，unsigned short int 和 long 变量分别占用 2 字节和 4 字节。

new 的返回值是一个内存地址。读者知道，内存地址被存储在指针中，因此应将 new 的返回值赋给一个指针。要在自由存储区中创建一个 unsigned short 变量，可以这样做：

```
unsigned short int * pPointer;  
pPointer = new unsigned short int;
```

当然，可以在声明指针的同时对其进行初始化，从而在一行代码中完成上述工作：

```
unsigned short int * pPointer = new unsigned short int;
```

无论采用哪种方式，pPointer 都将指向自由存储区中的一个 unsigned short int。可以像使用其他指向变量的指针那样使用它，将一个值赋给它指向的内存区域：

```
*pPointer = 72;
```

这条语句的含义是：将 72 放在 pPointer 指向的区域中或将 72 赋给 pPointer 指向的自由存储区中的区域。

#### 注意

如果 new 不能在自由存储区分配内存（毕竟内存是有限的资源），将引发异常（参见第 28 章）。

### 8.3.2 使用关键字 delete 归还内存

使用完内存区域后，必须将其归还给系统。为此，可以将 delete 应用于指针。delete 将内存归还给自由存储区。

请切记，使用 new 分配的内存不会被自动释放。如果指针变量指向自由存储区中的内存块，离开该指针的作用域时，该内存块不会被自动归还给自由存储区。相反，该内存块被视为已分配出去，同时由于该指针不再可用，您将无法访问该内存块。当指针为局部变量时将发生这样的情况。当函数返回时，在该函数中声明的指针将不在作用域中，从而丢失。使用 new 分配的内存不会被释放，而是变得不可用。

这被称为内存泄漏，因为在程序结束前，该内存块再也无法使用，就像从计算机中泄漏掉了一样。为防止内存泄漏，应将分配的内存归还给自由存储区。为此，可使用关键字 delete。例如：

```
delete pPointer;
```

删除指针时，实际上是释放了其地址存储在指针中的内存。这相当于将该指针指向的内存归还给

自由存储区。该指针仍然存在,可以重新给它赋值。程序清单 8.5 在堆中给一个变量分配内存,然后使用并删除它。

最常见的情况是,在构造函数中从堆中分配内存,在析构函数中释放这些内存。换句话说,您在构造函数中初始化指针,在对象被使用时为这些指针分配内存,并在析构函数中检查指针是否为空,如果不为空,则释放它们。

#### 警告

将 delete 用于指针时,它指向的内存将被释放。如果再次对该指针使用 delete,程序将崩溃!因此,删除指针后,应将其值设置为 0(空指针)。将 delete 用于空指针是安全的,例如:

```
Animal *pDog = new Animal; // allocate memory
delete pDog; // frees the memory
pDog = 0; // sets pointer to null
//...
delete pDog; // harmless
```

#### 程序清单 8.5 分配、使用和删除指针

```
1: // Listing 8.5
2: // Allocating and deleting a pointer
3: #include <iostream>
4: int main()
5: {
6:     using namespace std;
7:     int localVariable = 5;
8:     int * pLocal= &localVariable;
9:     int * pHeap = new int;
10:    *pHeap = 7;
11:    cout << "localVariable: " << localVariable << endl;
12:    cout << "pLocal: " << *pLocal << endl;
13:    cout << "pHeap: " << *pHeap << endl;
14:    delete pHeap;
15:    pHeap = new int;
16:    *pHeap = 9;
17:    cout << "pHeap: " << *pHeap << endl;
18:    delete pHeap;
19:    return 0;
20: }
```

#### ▼ 输出:

```
localVariable: 5
*pLocal: 5
*pHeap: 7
*pHeap: 9
```

#### ▼ 分析:

第 7 行声明并初始化了一个名为 localVariable 的局部变量,第 8 行声明一个名为 pLocal 的指针,并将其初始化为局部变量的地址。第 9 行声明了另外一个名为 pHeap 的指针,但将其初始化为 new int 的返回值,这从自由存储区中分配了可存储 int 值的内存块,该内存块可使用指针 pHeap 来访问。第 10 行将 7 存储到该内存块中。

第 11~13 行打印了几个值。第 11 行打印局部变量 localVariable 的值,第 12 行打印指针 pLocal 指向的值,第 13 行打印指针 pHeap 指向的值。正如所预期的,第 11 行和第 12 行打印的值相同。第 13 行表明,第 10 行所赋的值是可被访问的。

第 14 行通过调用 delete 将第 9 行分配的内存归还给自由存储区,这释放了内存,并将指针与内存脱离。现在,可以让 pHeap 指向任何其他内存。第 15 行和第 16 行重新给它赋值,第 17 行打印结果。第 18 行将内存归还给自由存储区。

虽然第 18 行是多余的(程序结束时将归还该内存块),但显式地释放是个不错的主意。如果程序需要修改或扩展,采取上述步骤将是有帮助的。

## 8.4 再谈内存泄漏

内存泄漏是最严重的问题之一，也是人们指责指针的原因之一。前面介绍了一种可能发生内存泄漏的情形。另一种可能导致内存泄漏的情形是，重新给指针赋值之前没有释放它原来指向的内存。请看下面的代码段：

```
1: unsigned short int * pPointer = new unsigned short int;
2: *pPointer = 72;
3: pPointer = new unsigned short int;
4: *pPointer = 84;
```

第1行创建了指针 pPointer，并将自由存储区中一个内存块的地址赋给它。第2行将 72 存储到这个内存块中。第3行重新将另一个内存块的地址赋给该指针。第3行将 84 存储到新的内存块中。原来的内存块（现在存储的值为 72）将不可用，因为指向该内存块的指针已被重新赋值。现在无法访问原来的内存块，在程序结束前也无法将其释放。

上述代码应写成这样：

```
1: unsigned short int * pPointer = new unsigned short int;
2: *pPointer = 72;
3: delete pPointer;
4: pPointer = new unsigned short int;
5: *pPointer = 84;
```

这样，第3行将释放 pPointer 原来指向的内存块。

### 注意

程序中的每个 new 都应该有对应的 delete。跟踪指针指向的内存区域并确保使用完毕后将其归还给自由存储区至关重要。

## 8.5 在自由存储区上创建对象

就像可以创建指向整型的指针一样，也可以创建指向任何数据类型（包括类）的指针。如果声明了一个 Cat 类，则可以声明一个指向 Cat 类的指针，并在自由存储区中实例化一个 Cat 对象，就像可以在堆栈中创建 Cat 对象一样。创建 Cat 指针的语法与创建 int 指针相同：

```
Cat *pCat = new Cat;
```

这将调用默认构造函数：不接受任何参数的构造函数。每当对象被创建（无论是在自由存储区还是堆栈中）都将调用构造函数。然而，需要注意的是，使用 new 创建对象时，不仅可以使默认构造函数，也可以使用任何构造函数。

## 8.6 删除自由存储区中的对象

将 delete 用于指向自由存储区中的对象的指针时，在释放内存之前将调用对象的析构函数。这给类提供了一个执行清理工作的机会（通常是释放从堆中分配而来的内存），就像从堆栈中删除对象一样。程序清单 8.6 演示了如何在自由存储区中创建和删除对象。

程序清单 8.6 在自由存储区中创建和删除对象

```
1: // Listing 8.6 - Creating objects on the free store
2: // using new and delete
3:
4: #include <iostream>
5:
6: using namespace std;
7:
8: class SimpleCat
9: {
10: public:
11:     SimpleCat();
12:     ~SimpleCat();
13: private:
```

```
14:     int itsAge;
15: };
16:
17: SimpleCat::SimpleCat()
18: {
19:     cout << "Constructor called. " << endl;
20:     itsAge = 1;
21: }
22:
23: SimpleCat::~SimpleCat()
24: {
25:     cout << "Destructor called. " << endl;
26: }
27:
28: int main()
29: {
30:     cout << "SimpleCat Frisky... " << endl;
31:     SimpleCat Frisky;
32:     cout << "SimpleCat *pRags = new SimpleCat..." << endl;
33:     SimpleCat * pRags = new SimpleCat;
34:     cout << "delete pRags... " << endl;
35:     delete pRags;
36:     cout << "Exiting, watch Frisky go... " << endl;
37:     return 0;
38: }
```

#### ▼ 输出:

```
SimpleCat Frisky...
Constructor called.
SimpleCat *pRags = new SimpleCat..
Constructor called.
delete pRags...
Destructor called.
Exiting, watch Frisky go...
Destructor called.
```

#### ▼ 分析:

第 8~15 行声明了一个简单类 `SimpleCat`。第 11 行声明了 `SimpleCat` 的构造函数，第 17~21 行是其定义。第 12 行声明了 `SimpleCat` 的析构函数，第 23~26 行是其定义。正如读者看到的，构造函数和析构函数都只是打印一条消息，让用户知道它们被调用了。

第 31 行将 `Frisky` 声明为一个常规局部变量，因此将在堆栈中创建它。这种创建导致构造函数被调用。第 33 行创建了一个被 `pRags` 指向的 `SimpleCat` 对象，然而，由于使用了指针，因此该对象将在堆中创建。同样，这将导致构造函数被调用。

第 35 行将 `delete` 用于指针 `pRags`。这将导致析构函数被调用，同时分配用于存储 `pRags` 指向的 `SimpleCat` 对象的内存被释放。当函数在第 38 行结束时，`Frisky` 不再在作用域中，因此其析构函数被调用。

## 8.7 迷途指针

有关指针的争论备受瞩目。这是因为在程序中由于指针引发的错误可能是最难发现和最难解决的。在 C++ 中，导致难以发现和解决的错误的罪魁祸首之一迷途 (stray) 指针。迷途指针也被称为失控 (wild) 指针或悬浮 (dangling) 指针，是将 `delete` 用于指针 (从而释放它指向的内存)，但没有将它设置为空时引发的。如果随后您在没有重新赋值的情况下使用该指针，后果将是不可预料的：程序崩溃算您走运。

这就如同 Acme Mail Order 公司变更了电话号码，但您仍去按原来绑定的按钮。这可能不会导致什么严重后果——也许这将拨打一个无人仓库中的电话。另一方面，这个号码也可能被重新分配给一个军工厂，您拨打电话可能引发爆炸，将整个城市摧毁。



总之，对指针使用 `delete` 后就不要再使用它。虽然这个指针仍指向原来的内存区域，但编译器可能已经将其他数据存储在里。不重新给这个指针赋值就再次使用它可能导致程序崩溃。更糟糕的是，程序可能表面上运行正常，但过不了几分钟就崩溃了。这被称为定时炸弹，可不是好玩的。为安全起见，删除指针后，把其设置空（0）。这样便解除了它的武装。程序清单 8.7 演示了如何创建迷途指针。

**警告**

这个程序故意创建了一个迷途指针。不要运行它，如果这样做，程序崩溃算您走运。

**程序清单 8.7 创建迷途指针**

```

1: // Listing 8.7 - Demonstrates a stray pointer
2:
3: typedef unsigned short int USHORT;
4: #include <iostream>
5:
6: int main()
7: {
8:     USHORT * pInt = new USHORT;
9:     *pInt = 10;
10:    std::cout << "pInt: " << *pInt << std::endl;
11:    delete pInt;
12:
13:    long * pLong = new long;
14:    *pLong = 90000;
15:    std::cout << "pLong: " << *pLong << std::endl;
16:
17:    *pInt = 20;    // uh oh, this was deleted!
18:
19:    std::cout << "pInt: " << *pInt << std::endl;
20:    std::cout << "pLong: " << *pLong << std::endl;
21:    delete pLong;
22:    return 0;
23: }
```

**▼ 输出:**

```

*pInt: 10
*pLong: 90000
*pInt: 20
*pLong: 65556
```

不要尝试去生成上面的输出。否则，如果您走运，输出将与此不同；如果您不走运，计算机将崩溃。

**▼ 分析:**

在重申一遍：不要运行该程序，因为它可能导致计算机死锁。第 8 行将 `pInt` 声明为一个 `UNSHORT` 指针，并将其指向使用 `new` 分配的内存。第 9 行将 10 存储到 `pInt` 指向的内存中，然后在第 10 行打印这个值。打印这个值后，对指针使用 `delete`。第 11 行被执行后，`pInt` 将成为一个迷途指针。

第 13 行声明了一个新的指针 `pLong`，它指向 `new` 分配的内存。第 14 行将 90000 存储到 `pLong` 指向的内存中，第 15 行打印这个值。

带来麻烦的是第 17 行，它将 20 存储到 `pInt` 指向的内存，但 `pInt` 不再指向任何合法的内存。第 11 行使用 `delete` 释放了 `pInt` 指向的内存，将一个值存储到该内存中无疑将带来灾难。

第 19 行打印 `pInt` 指向的值，当然应该是 20。第 20 行打印 `pLong` 指向的值，它突然变成了 65556。这提出了两个问题：

- 在没有动 `pLong` 的情况下，它指向的值怎么会变呢？
- 第 17 行使用 `pInt` 时将 20 存储到哪里了？

读者可能猜到了，这两个问题是相关的。第 17 行将一个值存储到 `pInt` 指向的内存中时，编译器将 20 存储到 `pInt` 原来指向的内存区域中。然而，由于第 11 行已经释放了这个内存块，编译器可以再次使用它。第 13 行创建指针 `pLong` 时，它指向的是 `pInt` 原来指向的内存块。（在有些计算机上，可能不

会发生这种情况，这取决于这些值保存在内存的什么位置)。将 20 赋给 pInt 原先指向的内存时，将覆盖 pLong 指向的值。这被称为“重踏指针”，它通常是使用迷途指针产生的不幸后果。

这种错误很难处理，因为被修改值与迷途指针毫无关联。pLong 指向的值被修改只是误用指针 pInt 的副作用。在大型程序中，这种错误很难查获。

#### 只是好玩

在程序清单 8.7 中，pLong 指向的内存中的值变成 65556 的过程如下：

1. 指针 pInt 指向某个内存块，并将 10 存储到该内存块中；
2. 将 delete 用于指针 pInt，这相当于告诉编译器，可以将该内存块用于存储其他东西，然后，指针 pLong 指向该内存块；
3. 将 90000 赋给 \*PLong。在这个例子中使用的计算机按字节交换顺序存储 4 字节值 90000 (00 01 5F 90)，因此它被存储为 5F 90 00 01；
4. 将 20 (十六进制表示为 00 14) 赋给 \*pInt。由于 pInt 仍然指向原来的地址，因此 pLong 的前两个字节被覆盖，变成了 00 14 00 00；
5. 打印 \*pLong 的值，将字节反转为正确的顺序 00 01 00 14，然后被转换为 DOS 值 65556。

#### FAQ

空指针和迷途指针的区别是什么？

答：将 delete 用于指针时，实际是让编译器释放内存，但指针本身依然存在。它现在是一个迷途指针。

如果接下来使用 myPtr = 0；，该迷途指针将变为空指针。

通常，如果对指针使用 delete 后再次对其使用 delete，后果将是不确定的，也就是说，任何事情都可能发生：如果幸运的话，程序可能会崩溃。但如果对空指针使用 delete，什么事情也不会发生，这样做是安全的。

使用迷途指针或空指针（如 myPtr=5；）是非法的，可能导致系统崩溃。如果是空指针，程序将崩溃，这是空指针相对于迷途指针的另一个优点。可预测的崩溃更可取，因为更容易调试。

## 8.8 使用 const 指针

声明指针时，可以在类型前或后使用关键字 const，也可在这两个位置都使用。例如，以下都是合法的声明：

```
const int * pOne;
int * const pTwo;
const int * const pThree;
```

然而，这些声明的含义是不同的：

- pOne 是一个指向整型常量的指针。它指向的值是不能修改的。
- pTwo 是一个指向整型的常量指针。它指向的值可以修改，但 pTwo 不能指向其他变量。
- pThree 是一个指向整型常量的常量指针。它指向的值不能修改，且这个指针也不能指向其他变量。

理解这些声明的技巧在于，查看关键字 const 右边来确定什么被声明为常量。如果该关键字的右边是类型，则值是常量；如果该关键字的右边是指针变量，则指针本身是常量。下面的代码有助于说明这一点：

```
const int * p1; // the int pointed to is constant
int * const p2; // p2 is constant, it can't point to anything else
```

应该	不应该
<p>如果对象不应被修改，则按引用传递它时应使用 <code>const</code> 进行保护。</p> <p>务必将指针设置为空，而不要让它未被初始化（悬浮）。</p>	<p>不要使用已被删除的指针。</p> <p>不要将指针删除多次。</p>

## 8.9 总结

指针提供了一种间接访问数据的强有力的手段。每个变量都有地址，可以通过地址运算符`&`来获得。地址可以存储在指针中。

要声明指针，可指出它指向的对象的类型，然后加上间接运算符`*`和指针名。应将指针初始化为指向一个对象或空`(0)`。要访问指针存储的地址处的值，可使用解除引用运算符`*`。可以声明 `const` 指针和指向 `const` 对象的指针。对于前者，不能给它重新赋值使之指向其他对象；而后者不能用于修改它指向的对象。

要在自由存储区中创建对象，可使用关键字 `new`，并将它返回的地址赋给一个指针。要释放指针指向的内存，将关键字 `delete` 用于该指针，但这样做不会销毁指针。因此，释放指针指向的内存后，必须给它重新赋值。

## 8.10 问与答

问：为什么指针如此重要？

答：指针重要的原因很多，其中包括指针可用于存储对象的地址和按引用传递参数。第 12 章将介绍指针在类多态中的应用。另外，很多操作系统和类库为您创建对象并返回指向它们的指针。

问：为什么要在自由存储区中声明对象？

答：自由存储区中的对象在函数返回后仍然存在。另外，在自由存储区中创建对象的功能让您能够在运行时决定需要多少个对象，而不是事先进行声明。这将在下一章深入讨论。

问：既然 `const` 对象限制了您可对它执行的操作，为什么要声明这样的对象呢？

答：作为程序员，您想让编译器帮助您查找错误。一种很难发现的严重错误是，函数以对调用函数来说不明显的方式修改对象。将对象声明为 `const` 可防止这种修改。

## 8.11 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 8.11.1 测验

1. 哪个运算符用于获得变量的地址？
2. 哪个运算符用于获得存储指针指向的地址处的值？

3. 什么是指针?
4. 指针指向的地址和存储在该地址中的值有何不同?
5. 间接运算符和地址运算符有何不同?
6. `const int * ptrOne` 和 `int * const ptrTwo` 有何不同?

## 8.11.2 练习

1. 下述声明的作用是什么?
  - a. `int * pOne;`
  - b. `int vTwo;`
  - c. `int * pThree = &vTwo;`
2. 如果有一个名为 `yourAge` 的 `unsigned short` 变量, 如何声明一个指针来操纵 `yourAge`?
3. 使用练习 2 中声明的指针将 50 赋给 `yourAge`。
4. 编写一个小程序, 在其中声明一个 `int` 变量和一个 `int` 指针, 并将 `int` 变量的地址赋给指针, 然后使用该指针来设置 `int` 变量的值。

5. 查错: 下面的代码有何错误?

```
#include <iostream>
using namespace std;
int main()
{
    int *pInt;
    *pInt = 9;
    cout << "The value at pInt: " << *pInt;
    return 0;
}
```

6. 查错: 下面的代码有何错误?

```
#include <iostream>
using namespace std;
int main()
{
    int SomeVariable = 5;
    cout << "SomeVariable: " << SomeVariable << endl;
    int *pVar = &SomeVariable;
    pVar = 9;
    cout << "SomeVariable: " << *pVar << endl;
    return 0;
}
```



## 第 9 章

# 使用引用

前一章介绍了如何使用指针来操纵自由存储区中的对象以及如何间接地访问这些对象；本章将介绍的引用几乎提供了指针的所有功能，但语法更简单。

在本章中，您将学习：

- 什么是引用
- 引用和指针的区别何在
- 如何创建和使用引用
- 引用的局限性是什么
- 如何按引用将值和对象传入和传出函数

### 9.1 什么是引用

引用是别名；创建引用时，您将其初始化为另一个对象（即目标）的名称。然后，引用将成为目标的另一个名称，对引用执行任何操作实际上都是针对目标的。

创建引用的方法是，首先给出目标对象的类型，然后加上引用运算符（&）、引用的名称、等号和目标对象的名称。

引用名可以是任何合法的变量名，但很多程序员喜欢在引用名中加上前缀 `r`。如果有一个名为 `someInt` 的 `int` 变量，可以编写如下代码来创建一个指向该变量的引用：

```
int &rSomeRef = someInt;
```

这条语句的含义是，`rSomeRef` 是一个指向 `int` 变量的引用，并被初始化为指向 `someInt`。引用和其他变量的区别在于，声明引用的同时必须对其进行初始化。如果创建引用时不给它赋值，将出现编译错误。程序清单 9.1 演示了如何创建和使用引用。

#### 注意

引用运算符（&）和地址运算符的符号相同，但它们并不是同一个运算符，虽然它们之间显然是相关的。

引用运算符之前的空格必不可少，而引用运算符和引用变量名之间的空格是可选的：

```
int &rSomeRef = someInt; // ok
int &rSomeRef = someInt; // ok
```

#### 程序清单 9.1 创建和使用引用

```
1: //Listing 9.1 - Demonstrating the use of references
2:
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     int intOne;
9:     int &rSomeRef = intOne;
10:
```



```
11:    intOne = 5;
12:    cout << "intOne: " << intOne << endl;
13:    cout << "rSomeRef: " << rSomeRef << endl;
14:
15:    rSomeRef = 7;
16:    cout << "intOne: " << intOne << endl;
17:    cout << "rSomeRef: " << rSomeRef << endl;
18:
19:    return 0;
20: }
```

▼ 输出:

```
intOne: 5
rSomeRef: 5
intOne: 7
rSomeRef: 7
```

▼ 分析:

第 8 行声明了一个局部 int 变量 intOne，第 9 行声明了一个 int 引用 rSomeRef，并将其初始化指向 intOne。正如前面指出的，如果声明了一个引用而没有对其进行初始化，将导致编译错误。必须对引用进行初始化。

第 11 行将 5 赋给 intOne。第 12 行和第 13 行打印 intOne 和 rSomeRef 的值，当然它们相同。

第 15 行将 7 赋给 rSomeRef。由于它是引用，是 intOne 的别名，因此 7 实际上被赋给了变量 intOne，如第 16 行和第 17 行的打印输出所示。

## 9.2 将地址运算符用于引用

读者知道，符号&可用于获得变量的地址以及声明引用。如果将地址运算符用于引用变量将如何呢？这将返回其指向的目标的地址。这就是引用的特征，引用是目标的别名。程序清单 9.2 演示了如何获取一个名为 rSomeRef 的引用变量的地址。

程序清单 9.2 获取引用的地址

```
1: //Listing 9.2 - Demonstrating the use of references
2:
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     int intOne;
9:     int &rSomeRef = intOne;
10:
11:     intOne = 5;
12:     cout << "intOne: " << intOne << endl;
13:     cout << "rSomeRef: " << rSomeRef << endl;
14:
15:     cout << "&intOne: " << &intOne << endl;
16:     cout << "&rSomeRef: " << &rSomeRef << endl;
17:
18:     return 0;
19: }
```

▼ 输出:

```
intOne: 5
rSomeRef: 5
&intOne: 0x3500
&rSomeRef: 0x3500
```

警告

最后两行为内存地址，这些内容随计算机上和每次运行而异，因此读者的输出可能不同。

## ▼ 分析:

同样, `rSomeRef` 也被初始化为指向 `intOne` 的引用, 但第 15 行和第 16 行打印这两个变量的地址, 它们相同。

在 C++ 中, 没有提供获取引用本身的地址的方法, 因为与指针或其他变量不同, 获取引用本身的地址毫无意义。引用在创建时被初始化, 此后它一直是目标的别名, 即使将地址运算符用于它时也是如此。

例如, 如果有一个 `President` 类, 您可能使用下面的语句来声明这个类的一个实例:

```
President George_Washington;
```

然后可以声明一个 `President` 引用, 并将其初始化为前面创建的对象:

```
President &FatherOfOurCountry = George_Washington;
```

在这种情况下, 只有一个 `President` 对象, 这两个标识符指的都是同一个类的同一个对象。对 `FatherOfOurCountry` 执行的任何操作都将针对 `George_Washington`。

请注意区分程序清单 9.2 中第 9 行的符号 `&` 和第 15 行及第 16 行的符号 `&`, 前者声明一个名为 `rSomeRef` 的 `int` 引用, 而后者返回 `int` 变量 `intOne` 和引用 `rSomeRef` 的地址。编译器知道如何根据上下文区分这两种用法。

## 注意

通常, 使用引用时不使用地址运算符, 引用就相当于目标变量。

## 不能给引用重新赋值

引用不能被重新赋值。即使是经验丰富的 C++ 程序员也搞不清给引用重新赋值将带来什么后果。引用变量总是其目标的别名。给引用重新赋值相当于给目标重新赋值, 程序清单 9.3 说明了这一点。

## 程序清单 9.3 给引用赋值

```
1: //Listing 9.3 - //Reassigning a reference
2:
3: #include <iostream>
4:
5: int main()
6: {
7:     using namespace std;
8:     int intOne;
9:     int &rSomeRef = intOne;
10:
11:     intOne = 5;
12:     cout << "intOne:    " << intOne << endl;
13:     cout << "rSomeRef:  " << rSomeRef << endl;
14:     cout << "&intOne:   " << &intOne << endl;
15:     cout << "&rSomeRef: " << &rSomeRef << endl;
16:
17:     int intTwo = 8;
18:     rSomeRef = intTwo; // not what you think!
19:     cout << "\nintOne:   " << intOne << endl;
20:     cout << "intTwo:    " << intTwo << endl;
21:     cout << "rSomeRef:  " << rSomeRef << endl;
22:     cout << "&intOne:   " << &intOne << endl;
23:     cout << "&intTwo:   " << &intTwo << endl;
24:     cout << "&rSomeRef: " << &rSomeRef << endl;
25:     return 0;
26: }
```

## ▼ 输出:

```
intOne:    5
rSomeRef:  5
&intOne:   0012FEDC
&rSomeRef: 0012FEDC
```

```

intOne:    8
intTwo:    8
rSomeRef:  8
&intOne:   0012FEDC
&intTwo:   0012FEE0
&rSomeRef: 0012FEDC

```

### ▼ 分析:

第 8 行和第 9 行声明了一个 int 变量和一个 int 引用。第 11 行将 5 赋给 int 变量, 第 12~15 行打印它们的值和地址。

第 17 行创建了一个新变量 intTwo, 并将它初始化为 8。在第 18 行, 程序员试图给 rSomeRef 重新赋值, 使其成为变量 intTwo 的别名, 但结果并非如此。实际情况是, rSomeRef 仍是变量 intOne 的别名, 这种赋值与下面的语句等价:

```
intOne = intTwo;
```

确实如此, 第 19~21 行打印 intOne 和 rSomeRef 的值时, 结果与 intTwo 相同。实际上, 从第 22~24 行打印的地址可知, rSomeRef 仍指向 intOne 而不是 intTwo。

应该	不应该
务必使用引用来创建对象的别名。 务必初始化所有的引用。	不要试图给引用重新赋值。 不要混淆地址运算符和引用运算符。

## 9.3 空指针和空引用

在指针没有被初始化或删除指针时, 应将它们赋为空 (0)。但引用并非如此, 因为必须在创建的同时将引用初始化为它指向的东西。

然而, 为让 C++ 可用于编写能够直接访问硬件的设备驱动程序、嵌入式系统和实时系统, 引用特定地址的能力很有用, 也必不可少。因此, 大多数编译器允许将引用初始化为空或数值, 仅当您试图在引用非法时使用对象时, 才会导致错误。

然而, 在常规编程中利用这种支持仍不是好主意。将使用了空引用的程序移植到其他计算机或编译器时, 可能出现难以查找的错误。

## 9.4 按引用传递函数参数

第 6 章介绍了函数的两个缺点: 参数按值传递, 返回语句只能返回一个值。

按引用将参数传递给函数可克服这两个缺点。在 C++ 中, 按引用传递参数有两种方式: 使用指针和使用引用。注意它们的不同: 使用指针按引用传递或使用引用按引用传递。

使用指针的语法和使用引用的语法不同, 但效果是相同的: 不是创建一个作用域为整个函数的副本, 而是相当于让函数能够访问原始对象。

按引用传递对象让函数能够修改被指向的对象。第 6 章介绍过, 传递给函数的参数存储在堆栈中。(使用指针或引用) 按引用将参数传递给函数时, 原始对象的地址而不是对象本身被存储到堆栈中。实际上, 在有些计算机上, 地址存储在寄存器中, 在堆栈中不存储任何东西。无论地址是存储在堆栈还是寄存器中, 编译器都知道如何获得原始对象, 修改是在原始对象而不是其副本中进行的。

第 6 章的程序清单 6.4 表明, 调用 swap() 函数并不会影响调用函数中的值。程序清单 9.4 再次列出了程序清单 6.4 中的代码, 以方便读者参考。

**程序清单 9.4 按值传递**

```
1: //Listing 9.4 - Demonstrates passing by value
2: #include <iostream>
3:
4: using namespace std;
5: void swap(int x, int y);
6:
7: int main()
8: {
9:     int x = 5, y = 10;
10:
11:     cout << "Main. Before swap, x: " << x << " y: " << y << endl;
12:     swap(x,y);
13:     cout << "Main. After swap, x: " << x << " y: " << y << endl;
14:     return 0;
15: }
16:
17: void swap (int x, int y)
18: {
19:     int temp;
20:
21:     cout << "Swap. Before swap, x: " << x << " y: " << y << endl;
22:
23:     temp = x;
24:     x = y;
25:     y = temp;
26:
27:     cout << "Swap. After swap, x: " << x << " y: " << y << endl;
28: }
```

**▼ 输出:**

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

**▼ 分析:**

该程序在 `main()` 中初始化了两个变量,然后将它们传递给函数 `swap()`,后者看似是交换这两个变量的值。在 `main()` 中再次检查这两个变量时,发现它们的值并没有变!

这里的问题是,参数 `x` 和 `y` 是按值传递给函数 `swap()` 的。也就是说,将在函数中创建局部副本。函数对局部副本进行修改,函数返回时,局部副本被丢弃,分配给它们的存储空间被释放。一种更好的方法是按引用传递 `x` 和 `y`,这样将修改原始变量而不是局部副本的值。

在 C++ 中,解决这个问题的方法有两种:将 `swap()` 的参数声明为指针,并让它指向原始值;传递指向原始值的引用。

### 9.4.1 使用指针让 `swap()` 管用

传递指针时,实际上传递的是对象的地址,这样函数便能够操纵存储在该地址处的值。要通过使用指针让函数 `swap()` 能够交换 `x` 和 `y` 的值,应将函数 `swap()` 声明为接受两个 `int` 指针参数。然后通过指针解除引用,访问 `x` 和 `y` 的值,并进行交换。程序清单 9.5 演示了这种想法。

**程序清单 9.5 通过使用指针来按引用传递**

```
1: //Listing 9.5 Demonstrates passing by reference
2: #include <iostream>
3:
4: using namespace std;
5: void swap(int *x, int *y);
6:
7: int main()
```

```
8: {
9:     int x = 5, y = 10;
10:
11:     cout << "Main. Before swap, x: " << x << " y: " << y << endl;
12:     swap(&x,&y);
13:     cout << "Main. After swap, x: " << x << " y: " << y << endl;
14:     return 0;
15: }
16:
17: void swap (int *px, int *py)
18: {
19:     int temp;
20:
21:     cout << "Swap. Before swap, *px: " << *px <<
22:         " *py: " << *py << endl;
23:
24:     temp = *px;
25:     *px = *py;
26:     *py = temp;
27:
28:     cout << "Swap. After swap, *px: " << *px <<
29:         " *py: " << *py << endl;
30:
31: }
```

#### ▼ 输出:

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, *px: 5 *py: 10
Swap. After swap, *px: 10 *py: 5
Main. After swap, x: 10 y: 5
```

#### ▼ 分析:

成功了! 在第 5 行, 函数 `swap()` 的原型被修改为接受两个 `int` 指针而不是 `int` 变量作为参数。第 12 行调用函数 `swap()` 时, 将 `x` 和 `y` 的地址作为参数传递给它。之所以知道传递的是地址, 是因为这里使用了地址运算符 (`&`)。

第 19 行在函数 `swap()` 中声明了一个局部变量 `temp`。`temp` 不必是指针, 它只是用于在函数的生命周期内存储 `*px` 的值 (调用函数中变量 `x` 的值)。函数返回后, 不再需要 `temp`。

第 24 行将 `temp` 赋给 `*px`。第 25 行将 `*py` 的值赋给 `*px`。第 26 行将 `temp` 的值 (即 `*px` 的原始值) 赋给 `*py`。

这样做的结果是, 调用函数中两个变量 (它们的地址被传递给函数 `swap()`) 的值确实被交换了。

## 9.4.2 使用引用来实现 `swap()`

上述程序虽然管用, 但函数 `swap()` 的语法比较繁琐, 这表现在两个方面。首先, 在函数 `swap()` 中需要对指针进行解除引用, 这很容易出错。如果没有对指针解除引用, 编译器仍允许将一个整数赋给指针, 但指针指向的地址将是错误的, 这也不好理解。其次, 需要在调用函数中传递变量的地址, 这使得 `swap()` 函数的内部工作原理对用户来说过于明显。

诸如 C++ 等面向对象编程语言的目标之一是, 让用户不必考虑函数的工作原理。使用指针传递参数将本应是被调用函数的责任转移到了调用函数。程序清单 9.6 使用引用重新编写了 `swap()` 函数。

#### 程序清单 9.6 使用引用重新编写的 `swap()`

```
1: //Listing 9.6 Demonstrates passing by reference
2: // using references!
3: #include <iostream>
4:
5: using namespace std;
6: void swap(int &x, int &y);
```



```
7:
8: int main()
9: {
10:     int x = 5, y = 10;
11:
12:     cout << "Main. Before swap, x: " << x << " y: "
13:         << y << endl;
14:
15:     swap(x,y);
16:
17:     cout << "Main. After swap, x: " << x << " y: "
18:         << y << endl;
19:
20:     return 0;
21: }
22:
23: void swap (int &rx, int &ry)
24: {
25:     int temp;
26:
27:     cout << "Swap. Before swap, rx: " << rx << " ry: "
28:         << ry << endl;
29:
30:     temp = rx;
31:     rx = ry;
32:     ry = temp;
33:
34:
35:     cout << "Swap. After swap, rx: " << rx << " ry: "
36:         << ry << endl;
37:
38: }
```

#### ▼ 输出:

```
Main. Before swap, x:5 y: 10
Swap. Before swap, rx:5 ry:10
Swap. After swap, rx:10 ry:5
Main. After swap, x:10, y:5
```

#### ▼ 分析:

和使用指针的范例一样，第 10 行声明了两个变量，并在第 12 行打印它们的值。第 15 行调用函数 `swap()`，但传递的是 `x` 和 `y`，而不是它们的地址。调用函数只是传递这两个变量。

函数 `swap()` 被调用时，程序跳到第 23 行执行，在这里参数为引用。第 27 行打印变量的值，但不需要使用任何特殊的运算符。它们都是原变量的别名，可以像原变量一样使用。

第 30~32 行交换参数的值，然后在第 35 行进行打印。程序跳回到调用函数中的第 17 行执行：打印 `main()` 中两个变量的值。由于函数 `swap()` 的参数被声明为引用，因此 `main()` 中的变量是按引用传递的，所以在 `main()` 中看到的也是修改后的值。从这个程序清单可知，引用在使用方面与常规变量一样方便和容易，同时具备指针的强大功能和按引用传递的能力。

## 9.5 返回多个值

正如本书前面讨论过的，函数只能返回一个值。如果需要从函数那里获得两个值，该如何呢？解决这种问题的方法之一是，按引用将两个对象传递给函数。然后，函数便可以将正确的值赋给这两个对象。由于按引用传递让函数能够修改原始对象，这相当于让函数能够返回两组信息。这种方法不求函数返回值，可以将返回值保留用于报告错误。

同样，这可以使用引用或指针来实现。程序清单 9.7 演示一个返回 3 个值的函数：其中两个为指针参数，一个为函数的返回值。

**程序清单 9.7 通过指针来返回值**

```
1: //Listing 9.7 - Returning multiple values from a function
2:
3: #include <iostream>
4:
5: using namespace std;
6: short Factor(int n, int* pSquared, int* pCubed);
7:
8: int main()
9: {
10:     int number, squared, cubed;
11:     short error;
12:
13:     cout << "Enter a number (0 - 20): ";
14:     cin >> number;
15:
16:     error = Factor(number, &squared, &cubed);
17:
18:     if (!error)
19:     {
20:         cout << "number: " << number << endl;
21:         cout << "square: " << squared << endl;
22:         cout << "cubed: " << cubed << endl;
23:     }
24:     else
25:         cout << "Error encountered!!" << endl;
26:     return 0;
27: }
28:
29: short Factor(int n, int *pSquared, int *pCubed)
30: {
31:     short Value = 0;
32:     if (n > 20)
33:         Value = 1;
34:     else
35:     {
36:         *pSquared = n*n;
37:         *pCubed = n*n*n;
38:         Value = 0;
39:     }
40:     return Value;
41: }
```

**▼ 输出:**

```
Enter a number (0-20): 3
number: 3
square: 9
cubed: 27
```

**▼ 分析:**

第 10 行将 `number`、`squared` 和 `cubed` 定义为 `int` 变量。第 14 行将用户输入的值赋给 `number`。第 16 行将 `number`、`squared` 和 `cubed` 的地址传递给函数 `Factor()`。

在第 32 行，函数 `Factor()` 检查第一个参数，这个参数是按值传递的。如果它大于 20（该函数能够处理的最大值），则将返回值 `Value` 设置为一个简单的错误值。函数 `Factor()` 的返回值用于指示错误信息，如果一切正常，将其设置为 0；在第 40 行，函数返回这个值。

实际需要的值（`number` 的平方和立方）并不是通过返回机制返回的，而是通过修改传递给函数的指针指向的值来返回的。

第 36 行和第 37 行设置指针指向的值，通过间接访问（通过将解除引用运算符 `*` 用于指针）将这些值赋给原来的变量。第 38 行将表示成功的值赋给变量 `Value`，第 40 行将其返回。

**提示**

由于按引用传递时不能控制对对象属性和方法的访问，因此应在让函数能完成其工作的情况下，将尽可能少的访问权限提供给函数。这有助于确保函数使用起来更安全，也更易于理解。

## 按引用返回值

虽然程序清单 9.7 管用，但如果使用引用而不是指针，程序将更容易理解和维护。程序清单 9.8 使用引用重新编写了该程序。

程序清单 9.8 还做了另一方面的改进：定义了一个枚举类型，使返回值更容易理解。该程序不是返回 0 或 1，而是通过使用枚举返回 SUCCESS 或 FAILURE。

**程序清单 9.8 使用引用重写程序清单 9.7**

```

1: //Listing 9.8
2: // Returning multiple values from a function
3: // using references
4: #include <iostream>
5:
6: using namespace std;
7:
8: enum ERR_CODE { SUCCESS, ERROR };
9:
10: ERR_CODE Factor(int, int&, int&);
11:
12: int main()
13: {
14:     int number, squared, cubed;
15:     ERR_CODE result;
16:
17:     cout << "Enter a number (0 - 20): ";
18:     cin >> number;
19:
20:     result = Factor(number, squared, cubed);
21:
22:     if (result == SUCCESS)
23:     {
24:         cout << "number: " << number << endl;
25:         cout << "square: " << squared << endl;
26:         cout << "cubed: " << cubed << endl;
27:     }
28:     else
29:         cout << "Error encountered!!" << endl;
30:     return 0;
31: }
32:
33: ERR_CODE Factor(int n, int &rSquared, int &rCubed)
34: {
35:     if (n > 20)
36:         return ERROR;    // simple error code
37:     else
38:     {
39:         rSquared = n*n;
40:         rCubed = n*n*n;
41:         return SUCCESS;
42:     }
43: }
```

**▼ 输出:**

```

Enter a number (0 - 20): 3
number: 3
square: 9
cubed: 27
```

### ▼ 分析:

程序清单 9.8 与程序清单 9.7 相比有两点不同: 枚举 `ERR_CODE` 使得第 36 行和第 41 行的错误报告以及第 22 行的错误处理更容易理解。

然而, 更大的变化是, 函数 `Factor()` 现在被声明为接受引用而不是指针作为参数。这使得对这些参数的操纵更简单, 更容易理解。

## 9.6 按引用传递以提高效率

按值将对象传递给函数时, 都将创建该对象的一个副本; 而按值从函数返回一个对象时, 将创建另一个副本。对于小型对象, 如内置的整数值, 这样的开销是微不足道的。

然而, 对于用户定义的大型对象 (如结构或类对象), 复制的开销可能很高。用户定义的对象在堆栈中占据的空间是其所有的成员变量所占空间的总和, 而这些成员变量本身又可能是用户创建的对象, 通过在堆栈中复制来传递如此庞大的结构, 无论是在性能方面还是在内存占用方面都是非常昂贵的。

还有另一种开销。每当创建这些临时副本时, 都要调用一个特殊的构造函数: 复制构造函数。有关复制构造函数的工作原理以及如何创建自己的复制构造函数, 将在本书后面介绍。就现在而言, 读者只需知道每次在堆栈中创建临时副本时都将调用复制构造函数即可。

函数返回时, 临时对象将被销毁, 这将调用对象的析构函数。如果函数按值返回对象, 将需要创建和销毁该对象的一个副本。

对于大型的对象, 调用构造函数和析构函数在速度和内存方面的开销都可能很大。为说明这一点, 程序清单 9.9 创建了一个简单的用户定义的对象: `SimpleCat`。实际的对象可能更大, 开销也可能更高, 但使用它足以说明复制调用构造函数和析构函数的调用频率。由于本书还未介绍类, 因此不要将注意力放在语法上, 而将重点放在结果上, 以理解按引用传递如何减少函数调用。

程序清单 9.9 按引用传递对象

```
1: //Listing 9.9 - Passing pointers to objects
2:
3: #include <iostream>
4:
5: using namespace std;
6: class SimpleCat
7: {
8:     public:
9:         SimpleCat ();           // constructor
10:        SimpleCat(SimpleCat&);   // copy constructor
11:        ~SimpleCat();           // destructor
12: };
13:
14: SimpleCat::SimpleCat()
15: {
16:     cout << "Simple Cat Constructor..." << endl;
17: }
18:
19: SimpleCat::SimpleCat(SimpleCat&)
20: {
21:     cout << "Simple Cat Copy Constructor..." << endl;
22: }
23:
24: SimpleCat::~~SimpleCat()
25: {
26:     cout << "Simple Cat Destructor..." << endl;
27: }
28:
29: SimpleCat FunctionOne (SimpleCat theCat);
30: SimpleCat* FunctionTwo (SimpleCat *theCat);
31:
```

```

32: int main()
33: {
34:     cout << "Making a cat..." << endl;
35:     SimpleCat Frisky;
36:     cout << "Calling FunctionOne..." << endl;
37:     FunctionOne(Frisky);
38:     cout << "Calling FunctionTwo..." << endl;
39:     FunctionTwo(&Frisky);
40:     return 0;
41: }
42:
43: // FunctionOne, passes by value
44: SimpleCat FunctionOne(SimpleCat theCat)
45: {
46:     cout << "Function One. Returning..." << endl;
47:     return theCat;
48: }
49:
50: // functionTwo, passes by reference
51: SimpleCat* FunctionTwo (SimpleCat *theCat)
52: {
53:     cout << "Function Two. Returning..." << endl;
54:     return theCat;
55: }

```

#### ▼ 输出:

```

Making a cat...
Simple Cat Constructor...
Calling FunctionOne...
Simple Cat Copy Constructor...
Function One. Returning...
Simple Cat Copy Constructor...
Simple Cat Destructor...
Simple Cat Destructor...
Calling FunctionTwo...
Function Two. Returning...
Simple Cat Destructor...

```

#### ▼ 分析:

程序清单 9.10 创建了一个 SimpleCat 对象，然后调用两个函数。第一个函数按值接受一个 Cat 对象，然后按值返回它。第二个函数接受一个对象指针而不是对象本身作为参数，并返回一个指向对象的指针。

第 6~12 行声明了一个非常简单的 SimpleCat 类。构造函数、复制构造函数和析构函数打印一条消息，以便它们被调用时读者能够知道。

在第 34 行，main() 函数打印一条消息，即输出中的第一行。第 35 行实例化一个 SimpleCat 对象。这将导致构造函数被调用，输出中的第 2 行就是构造函数打印的消息。

在第 36 行，main() 函数指出将调用 FunctionOne，这是输出中的第 3 行。由于调用 FunctionOne() 时按值传递了 SimpleCat 对象，因此将在堆栈中创建该 SimpleCat 对象的一个副本，将其作为被调用函数的局部对象。这导致复制构造函数被调用，生成输出中的第 4 行。

然后，程序跳到被调用函数中的第 46 行执行：打印一条消息，这是输出中的第 5 行。然后函数返回，并按值返回 SimpleCat 对象，这将创建该对象的另一个副本，导致复制构造函数被调用，生成输出中的第 6 行。

函数 FunctionOne() 的返回值没有被赋给任何对象，因此为返回而创建的临时对象被丢弃，这导致析构函数被调用，生成输出中的第 7 行。由于 FunctionOne() 已经结束，其局部副本不再在作用域中，因此将被销毁，导致析构函数被调用，生成输出中的第 8 行。

程序返回到 main() 继续执行，并调用 FunctionTwo() 函数，但按引用传递参数。这不会创建对象副本，因此没有输出。FunctionTwo() 打印位于输出中第 10 行的消息，然后按引用返回 SimpleCat 对象，因此不会导致调用构造函数和析构函数被调用。



最后，程序结束，Frisky 不再在作用域中，导致最后一次调用析构函数并打印输出中的最后一行。调用 FunctionOne( ) 时，由于按值传递 Frisky，导致复制构造函数和析构函数被调用两次；调用 FunctionTwo( ) 时没有导致复制构造函数和析构函数被调用。

### 9.6.1 传递 const 指针

虽然给函数 FunctionTwo( ) 传递指针的效率更高，但这样做也是危险的。函数 FunctionTwo( ) 并不打算对传递给它的 SimpleCat 对象进行修改，但仍获得了该对象的地址。这就使原始对象暴露在被修改的危险之中，失去了按值传递提供的保护。

按值传递就像将作品的照片而不是实物交给博物馆，在这张照片上做任何改动都不会损害原物。按引用传递就像将家里的地址告诉博物馆，并邀请客人来家中观看实物。

解决这个问题的方法是，传递一个指向 const SimpleCat 对象的指针。这样做可防止对 SimpleCat 对象调用任何非 const 方法，从而防止对象被改变。

传递 const 引用让客人能够看到原物，但不允许做任何修改。程序清单 9.10 说明了这种思想。

程序清单 9.10 传递指向 const 对象的指针

```
1: //Listing 9.10 - Passing pointers to objects
2:
3: #include <iostream>
4:
5: using namespace std;
6: class SimpleCat
7: {
8:     public:
9:         SimpleCat();
10:        SimpleCat(SimpleCat&);
11:        ~SimpleCat();
12:
13:        int GetAge() const { return itsAge; }
14:        void SetAge(int age) { itsAge = age; }
15:
16:    private:
17:        int itsAge;
18: };
19:
20: SimpleCat::SimpleCat()
21: {
22:     cout << "Simple Cat Constructor..." << endl;
23:     itsAge = 1;
24: }
25:
26: SimpleCat::SimpleCat(SimpleCat&)
27: {
28:     cout << "Simple Cat Copy Constructor..." << endl;
29: }
30:
31: SimpleCat::~~SimpleCat()
32: {
33:     cout << "Simple Cat Destructor..." << endl;
34: }
35:
36: const SimpleCat * const FunctionTwo
37:     (const SimpleCat * const theCat);
38:
39: int main()
40: {
41:     cout << "Making a cat..." << endl;
42:     SimpleCat Frisky;
43:     cout << "Frisky is ";
44:     cout << Frisky.GetAge();
45:     cout << " years old" << endl;
46:     int age = 5;
```

```

47:     Frisky.SetAge(age);
48:     cout << "Frisky is " ;
49:     cout << Frisky.GetAge();
50:     cout << " years old" << endl;
51:     cout << "Calling FunctionTwo..." << endl;
52:     FunctionTwo(&Frisky);
53:     cout << "Frisky is " ;
54:     cout << Frisky.GetAge();
55:     cout << " years old" << endl;
56:     return 0;
57: }
58:
59: // functionTwo, passes a const pointer
60: const SimpleCat * const FunctionTwo
61:     (const SimpleCat * const theCat)
62: {
63:     cout << "Function Two. Returning..." << endl;
64:     cout << "Frisky is now " << theCat->GetAge();
65:     cout << " years old " << endl;
66:     // theCat->SetAge(8);    const!
67:     return theCat;
68: }

```

#### ▼ 输出:

```

Making a cat...
Simple Cat constructor...
Frisky is 1 years old
Frisky is 5 years old
Calling FunctionTwo...
FunctionTwo. Returning...
Frisky is now 5 years old
Frisky is 5 years old
Simple Cat Destructor...

```

#### ▼ 分析:

在 SimpleCat 类中添加了两个存取器函数:第 13 行的 GetAge() 和第 14 行的 SetAge(), 其中 GetAge() 是一个 const 方法, 而 SetAge() 不是。还在第 17 行添加了成员变量 itsAge。

构造函数、复制构造函数和析构函数仍被定义为打印相应的消息。然而, 复制构造函数永远不会被调用, 因为对象是按引用传递的, 不会创建对象副本。第 42 行创建一个对象, 从第 43 行开始打印它的默认年龄。

第 47 行使用存取器函数 SetAge() 设置 itsAge 的值, 第 48 行打印结果。在这个程序中, 没有使用 FunctionOne(), 但调用了 FunctionTwo()。FunctionTwo() 有细微的变化, 第 36 行将其参数和返回值声明为指向 const 对象的 const 指针。

由于参数和返回值仍是按引用传递的, 因此不需要创建对象副本, 也就不会调用复制构造函数。然后, 由于在函数 FunctionTwo() 中, 被指向的对象为 const, 因此不能调用非 const 方法 SetAge()。如果不将调用 SetAge() 的第 66 行注释掉, 程序将不能通过编译。

注意, 在 main() 函数中创建的对象并不是 const 的, 因此 Frisky 能够调用函数 SetAge()。这个非 const 对象的地址被传递给函数 FunctionTwo(), 但由于该函数的原型将指针声明为指向 const 对象的 const 指针, 因此该对象被视为 const 的。

## 9.6.2 用引用代替指针

程序清单 9.10 解决了需要创建副本的问题, 从而减少了对复制构造函数和析构函数的调用。它使用指向 const 对象的 const 指针, 因此也解决了在函数中可能修改对象的问题。然而, 这有些繁琐, 因为传递给函数的是指向对象的指针。

由于对象不可能为空, 如果传递引用而不是指针, 则在函数中处理起来将更方便。程序清单 9.11

说明了这一点。

程序清单 9.11 传递指向对象的引用

```
1: //Listing 9.11 - Passing references to objects
2:
3: #include <iostream>
4:
5: using namespace std;
6: class SimpleCat
7: {
8:     public:
9:         SimpleCat();
10:        SimpleCat(SimpleCat&);
11:        ~SimpleCat();
12:
13:        int GetAge() const { return itsAge; }
14:        void SetAge(int age) { itsAge = age; }
15:
16:     private:
17:         int itsAge;
18: };
19:
20: SimpleCat::SimpleCat()
21: {
22:     cout << "Simple Cat Constructor..." << endl;
23:     itsAge = 1;
24: }
25:
26: SimpleCat::SimpleCat(SimpleCat&)
27: {
28:     cout << "Simple Cat Copy Constructor..." << endl;
29: }
30:
31: SimpleCat::~~SimpleCat()
32: {
33:     cout << "Simple Cat Destructor..." << endl;
34: }
35:
36: const SimpleCat & FunctionTwo (const SimpleCat & theCat);
37:
38: int main()
39: {
40:     cout << "Making a cat..." << endl;
41:     SimpleCat Frisky;
42:     cout << "Frisky is " << Frisky.GetAge() << " years old" << endl;
43:     int age = 5;
44:     Frisky.SetAge(age);
45:     cout << "Frisky is " << Frisky.GetAge() << " years old" << endl;
46:     cout << "Calling FunctionTwo..." << endl;
47:     FunctionTwo(Frisky);
48:     cout << "Frisky is " << Frisky.GetAge() << " years old" << endl;
49:     return 0;
50: }
51:
52: // functionTwo, passes a ref to a const object
53: const SimpleCat & FunctionTwo (const SimpleCat & theCat)
54: {
55:     cout << "Function Two. Returning..." << endl;
56:     cout << "Frisky is now " << theCat.GetAge();
57:     cout << " years old " << endl;
58:     // theCat.SetAge(8);    const!
59:     return theCat;
60: }
```

▼ 输出:

```
Making a cat...
Simple Cat constructor...
Frisky is 1 years old
Frisky is 5 years old
Calling FunctionTwo...
```

```
FunctionTwo. Returning...
Frisky is now 5 years old
Frisky is 5 years old
Simple Cat Destructor...
```

▼ 分析:

该程序的输出与程序清单 9.10 相同。唯一明显的变化是，现在函数 `FunctionTwo()` 接受一个指向 `const` 对象的引用作为参数，并返回一个这样的引用。同样，使用引用比使用指针更简单，但在内存节省和效率方面与指针相同，同时可以使用 `const` 来提供安全。

const 引用

C++程序员对“指向 `SimpleCat` 对象的 `const` 引用”和“指向 `const SimpleCat` 对象的引用”通常是不加以区分的。由于不能给引用重新赋值使其指向另一个对象，因此它们总是 `const` 的。如果将关键字 `const` 用于引用，将使指向的对象为 `const` 的。

9.7 何时使用引用和指针

相对于指针来说，经验丰富的 C++程序员更喜欢使用引用。引用不但更清晰，更容易使用，而且能够更好地隐藏信息，如前面的例子所示。

然而，引用不能被重新赋值。如果需要首先指向一个对象，然后指向另一个，则必须使用指针。引用不能为空，因此如果对象可能为空，则绝对不能使用引用，而必须使用指针。

第二种情况的一个例子是使用 `new` 运算符来创建对象。如果 `new` 不能在自由存储区中分配内存，将返回一个空指针。由于引用不能为空，因此除非已经确定内存不为空，否则不要将引用初始化为指向该内存。下面的例子演示了如何处理这种情况：

```
int *pInt = new int;
if (pInt != NULL)
    int &rInt = *pInt;
```

在这个例子中，声明了一个 `int` 指针 `pInt`，并将其初始化为指向 `new` 运算符返回的内存。然后测试 `pInt` 的地址，如果不为空，则对 `pInt` 解除引用（对 `int` 指针解除引用的结果是为 `int` 对象），并将 `rInt` 初始化为指向这个对象。这样，`rInt` 就成了运算符 `new` 返回的 `int` 对象的别名。

应该	不应该
尽可能按引用传递参数。 尽可能使用 <code>const</code> 来保护引用和指针。	在可以使用引用时不要使用指针。 不要试图给引用重新赋值，使之指向另一个变量，这是不可能的。

9.8 混合使用引用和指针

在同一个函数参数列表中同时声明指针和引用以及按值传递对象是完全合法的。下面是一个这样的例子：

```
Cat * SomeFunction (Person &theOwner, House *theHouse, int age);
```

该声明指出，函数 `SomeFunction()` 接受 3 个参数。第 1 个是指向 `Person` 对象的引用，第 2 个是指向 `House` 对象的指针，第 3 个是一个 `int` 变量。该函数返回了一个指向 `Cat` 对象的指针。

关于在声明这些变量时，将引用运算符（&）和间接访问运算符（\*）放在什么地方存在较大的争论。以下声明引用的方式都是合法的：

```
1: Cat& rFrisky;
2: Cat & rFrisky;
3: Cat  &rFrisky;
```

空白被完全忽略了，因此任何有空格的地方都可以放置任意数目的空格、制表符和换行。

撇开自由表达的问题不谈，哪种方式最好呢？以下是认为各种方式最好的理由。

第一种方式最好的理由是，`rFrisky` 是一个变量，其名称为 `rFrisky`，类型为指向 `Cat` 对象的引用。因此这种观点认为，`&` 应和类型放在一起。

反对者认为，类型应为 `Cat`。`&` 是声明的一部分，声明包括变量名和 `&`。更重要的是，将 `&` 和 `Cat` 放在一起导致下面这样的错误：

```
Cat& rFrisky, rBoots;
```

如果不注意，可能认为 `rFrisky` 和 `rBoots` 都是指向 `Cat` 对象的引用，但是您错了。这行代码的实际意思是，`rFrisky` 是一个 `Cat` 引用，而 `rBoots` 不是引用（虽然名称中包含前缀 `r`），而是一个 `Cat` 对象。这行代码应这样书写：

```
Cat    &rFrisky, rBoots;
```

对于这种写法，反对者认为，就不应像上面这样将引用声明和变量声明混在一起。正确的方式如下：

```
Cat& rFrisky;  
Cat  boots;
```

最后，很多程序员选择置身于这种争论之外，将 `&` 放在类型和名称中间，如第二种方式所示。

当然，这里对引用运算符（`&`）的所有讨论也适用于间接运算符（`*`）。重要的在于，对于哪种方式是正确的存在不同看法。选择一种适合于自己的风格，并在同一个程序保持一致。代码清晰仍然是我们的目标。

#### 注意

很多程序员喜欢使用下述声明引用和指针的方式：

- 将 `&` 和 `*` 放在中间，两边各加一个空格；
- 决不要在同一行中同时声明引用、指针和变量。

## 9.9 返回指向不在作用域中的对象的引用

C++ 程序员学习按引用传递后，常常会过度迷恋，进而滥用它。别忘了，引用是其他对象的别名。将引用传入或传出函数时，务必这样自问：引用指向的对象是什么？每次使用它时它是否存在？

程序清单 9.12 说明了返回指向不再存在的对象的引用的危险性。

程序清单 9.12 返回指向不存在的对象的引用

```
1: #include <iostream>  
2:  
3: int& GetInt ();  
4:  
5: int main()  
6: {  
7:     int & rInt = GetInt ();  
8:     std::cout << "rInt = " << rInt << std::endl;  
9:  
10:    return 0;  
11: }  
12:  
13: int & GetInt ()  
14: {  
15:     int nLocalInt = 25;  
16:  
17:     return nLocalInt;  
18: }
```

#### ▼ 输出：

Compile error: Attempting to return a reference to a local object!



警告

这个程序在 Borland 编译器中不能通过编译，但在微软编译器中能够通过编译，只是可能有警告。然而，需要指出的是，这是一种糟糕的编码习惯。

▼ 分析：

在函数 `GetInt()` 内部声明了一个类型为 `int` 的局部变量，并将其初始化为 25，然后按引用返回该变量。有些编译器很聪明，能够发现这种错误，进而不允许您运行该程序。其他编译器允许您运行该程序，但结果是不确定的。

函数 `GetInt()` 返回时，局部变量 `nLocalInt` 将被销毁。该函数返回的引用是一个并不存在的变量的别名，这很糟糕。

返回指向堆/自由存储区中对象的引用

您可能试图让函数 `GetInt()` 在堆中创建变量 `nLocalInt`，以解决程序清单 9.12 中的问题。这样，当函数 `GetInt()` 返回时 `nLocalInt` 仍存在。

这种解决方法存在的问题是：使用完 `nLocalInt` 后，如何释放分配给它的内存？程序清单 9.13 说明了这种问题。

程序清单 9.13 内存泄漏

```
1: #include <iostream>
2:
3: int& GetInt ();
4:
5: int main()
6: {
7:     int & rInt = GetInt ();
8:     std::cout << "rInt = " << rInt << std::endl;
9:
10:    return 0;
11: }
12:
13: int & GetInt ()
14: {
15:     // Instantiate an integer object on the free store / heap
16:     int* pInteger = new int (25);
17:
18:     return *pInteger;
19: }
```

▼ 输出：

rInt = 25

警告

该程序能够通过编译和链接，且看起来运行也正常，但它是一颗定时炸弹。

▼ 分析：

在第 13~19 行，对函数 `GetInt()` 进行了修改，使其不再返回一个指向局部变量的引用。第 16 行从自由存储区中分配内存，并将其地址赋给一个指针，然后对指针 `pInteger` 解除引用并按引用返回该指针指向的对象。第 7 行将函数 `GetInt()` 的返回值赋给一个引用，然后第 8 行使用该引用来获得整数值并打印它。

到目前为止，一切顺利。但如何释放被占用的内存呢？不能引用使用 `delete` 运算符，一种解决方案是创建另一个指针，并将其初始化为从 `rInt` 获得的地址。这样确实可以释放内存，从而避免内存泄漏，但存在一个小问题：此后 `rInt` 将指向什么呢？正如前面指出的，引用必须始终是一个实际对象的别名；如果指向一个空对象（就像现在这样），程序将是非法的。因此，应尽可能避免采用这种解决方案，因为有更简单、更清晰的解决方案。

注意

使用指向空对象的引用的程序也许能够通过编译，但它是非法的，结果是不可预料的，对于这一点如何强调都不过分。

对于这个问题，存在下面 3 种解决方案：

```
int GetInt ();  
int* GetInt ();  
void GetInt (int & nInt);
```

应该	不应该
在必须按值传递参数时务必这样做。 在必须按值返回时务必这样做。	如果被引用的对象可能位于作用域外，不要按引用传递。 不要失去内存是何时在什么地方分配的线索，以确保内存得到释放。

## 9.10 总结

本章介绍引用并将其同指针进行了比较。必须对引用进行初始化，使之指向一个现有的对象；同时不能给引用重新赋值，使之指向其他对象。对引用的任何操作实际上针对的都是引用的目标对象。有关这一点的证据是，将地址运算符用于引用时，返回的是它指向的目标对象的地址。

与按值传递相比，按引用传递对象的效率更高。另外，按引用传递让被调用的函数能够修改调用函数传入的实参。

函数的参数和返回值可以按引用传递，这可以使用指针或引用来实现。

读者学习了如何使用指向 const 对象的指针和 const 引用在函数之间安全地传递值，同时获得按引用传递的高效率。

## 9.11 问与答

问：既然指针具备引用的所有功能，为什么还要使用引用？  
答：引用更容易使用和理解。间接被隐藏，不需要重复地对变量解除引用。

问：既然引用更容易，为什么还要使用指针呢？  
答：引用不能为空，也不能被重新赋值。指针提供了更大的灵活性，但使用起来更难些。

问：为什么要从函数按值返回？  
答：如果被返回的对象是局部的，必须按值返回，否则将返回指向不存在的对象的引用。

问：鉴于按引用返回的危险性，为什么不总是按值返回呢？  
答：按引用返回的效率 high 得多。这样做不仅可以节省内存，程序的运行速度也更快。

## 9.12 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 9.12.1 测验

1. 引用和指针之间有什么区别?
2. 什么时候必须使用指针而不是引用?
3. 如果没有足够的内存来创建新对象, new 将返回什么?
4. 什么是 const 引用?
5. 按引用传递和传递引用之间有何区别?
6. 下面哪种声明引用的方式是正确的?
  - a. `int& myRef = myInt;`
  - b. `int & myRef = myInt;`
  - c. `int &myRef = myInt;`

### 9.12.2 练习

1. 编写一个程序, 声明一个 int、一个 int 引用和一个 int 指针, 然后使用指针和引用来操纵 int 变量的值。

2. 编写一个程序, 声明一个指向 const int 变量的 const 指针。将该指针初始化为指向 int 变量 varOne, 并将 6 赋给 varOne。使用该指针将 7 赋给 varOne。创建另一个 int 变量 varTwo, 给指针重新赋值, 使之指向为 varTwo。暂不要编译这个程序。

3. 编译练习 2 中的程序。将出现什么错误? 什么警告?

4. 编写一个生成迷失 (stray) 指针的程序。

5. 修复练习 4 中程序的问题。

6. 编写一个导致内存泄漏的程序。

7. 修复练习 6 中程序的问题。

8. 查错: 下面的程序有什么错误?

```

1:  #include <iostream>
2:  using namespace std;
3:  class CAT
4:  {
5:  public:
6:      CAT(int age) { itsAge = age; }
7:      ~CAT(){}
8:      int GetAge() const { return itsAge;}
9:  private:
10:     int itsAge;
11: };
12:
13: CAT & MakeCat(int age);
14: int main()
15: {
16:     int age = 7;
17:     CAT Boots = MakeCat(age);
18:     cout << "Boots is " << Boots.GetAge()
19:          << " years old" << endl;
20:     return 0;
21: }
22:
23: CAT & MakeCat(int age)
24: {
25:     CAT * pCat = new CAT(age);
26:     return *pCat;
27: }
```

9. 修复练习 8 中程序的问题。





## 第二部分

# 面向对象编程和 C++ 基础

第 10 章 类和对象

第 11 章 实现继承

第 12 章 多态

第 13 章 运算符类型与运算符重载

第 14 章 类型转换运算符

第 15 章 宏和模板简介



# 第 10 章

## 类和对象

类扩展了 C++ 内置的功能，可帮助用户表示并解决复杂的实际问题。

在本章中，您将学习：

- 什么是类和对象
- 如何定义新类并创建其对象
- 什么是成员函数和成员数据
- 什么是构造函数？如何使用它们

### 10.1 C++ 是面向对象的吗

C++ 的前身 C 语言一度是世界上最流行的用于商业软件开发的程序设计语言，它被用于开发操作系统（如 UNIX 操作系统）、进行实时编程（机器、设备和电子控制），随后才被用作一种常规编程语言。其目标是提供一种更容易、更安全的与硬件交互的编程方式。

C 是作为一种介于高级商业应用程序语言（如 COBOL）与低级、高性能、很难使用的汇编语言之间的中间语言而开发的。C 语言采用结构化编程，在这种编程方式中，问题被分解成较小的名为过程的重重复活动单元，而数据被组织成名为结构的包。

然而，诸如 Smalltalk 和 CLU 等研究用语言开拓了一个新方向：面向对象，将存储在结构中的数据与过程的功能组合成一个单元——对象。

现实世界中充斥着对象（汽车、狗、树、云、花）；每个对象都有其特征（速度快、友好、棕色、蓬松、可爱）；大多数对象都有行为（移动、叫、长大、下雨、枯萎）。通常，人们不会考虑汽车的规格以及如何操纵这些规格，而是将汽车视为一种有特定外观和行为的对象。在计算机领域描述现实世界中的对象时，情况也如此。

我们在 21 世纪初编写的程序要比在 20 世纪末编写的程序复杂得多。使用过程性语言编写的程序通常难以管理和维护，且扩展的费用高昂。图形用户界面、Internet、数字和无线电话以及大量新技术的出现，极大地增加项目的复杂性，同时用户对用户界面质量的期望也在增加。

面向对象软件开发提供了帮助应对这种软件开发挑战的工具。虽然对于复杂的软件开发，并不存在什么灵丹妙药，但面向对象编程语言将数据结构和操纵数据的方法紧密地联系在一起，更符合人类（程序员和客户）的思维方式，可促进交流，改善交付的软件的质量。在面向对象编程中，考虑的不再是数据结构和操纵它们的函数，而是对象，它们就像现实世界中的物体：有特定的外观和行为。

创建 C++ 旨在面向对象编程和 C 语言之间架起一座桥梁，其目标是提供一种面向对象设计的快速商用软件开发平台，并将重点放在高性能上。接下来读者将更详细地了解到 C++ 是如何实现其目标的。



## 10.2 创建新类型

编写程序通常旨在解决实际问题，如跟踪员工记录或模拟供暖系统的工作原理。虽然只使用数字和字符编写的程序也可以解决复杂的问题，但如果能够创建您谈论的对象，则解决复杂的大型问题将容易得多。换句话说，如果能创建表示房间、热传感器、温度调节装置和锅炉的变量，则模拟供热系统工作原理将容易得多。这些变量与现实情况联系越紧密，程序编写起来越容易。

前面介绍了大量的变量类型，包括 `unsigned int` 变量和 `char` 变量。类型指出了有关变量的很多信息。例如，如果 `Height` 和 `Width` 被声明为 `unsigned short` 变量，您将知道其中每个变量都可以存储一个 0~65 535 的数，这里假设 `unsigned short` 占用 2 字节。这就是 `unsigned short` 的含义，如果在这些变量中存储其他的数据，将导致错误。不能将您的姓名存储在 `unsigned short` 变量中，也不应试图这样做。

通过将 `Height` 和 `Width` 声明为 `unsigned short`，您知道可以将它们相加，并将结果赋给另一个数值变量。

变量类型提供了如下信息：

- 变量在内存中的长度；
- 变量可存储什么样的信息；
- 可对变量执行什么样的操作。

在传统的语言（如 C 语言）中，类型被内置到语言中。在 C++ 中，程序员可以通过创建所需的任何类型来扩展该语言，每种新类型都可以有与内置类型相同的功能。

### 使用结构创建类型的缺点

通过将相关变量组合成结构，提供了在 C 语言中增加新类型的功能。通过使用 `typedef` 语句，可让结构称为一种新的数据类型。

然而，这种功能有如下缺点：

- 结构和操纵它们的函数不是一个有机的整体——只能通过阅读库的头文件，并使用新类型作为参数进行查找，才能找到函数；
- 针对结构的相关函数组的行为进行协调非常困难，因为结构中的任何数据可能在任何时候被任何程序逻辑修改，无法防止结构数据不受干扰；
- 内置的运算符不适用于结构——不能使用 `+` 将两个结构相加，即使这可能是一种最自然的表示问题解决方案的方式（例如，当每个结构表示要组合在一起的一串文本时）。

## 10.3 类和成员简介

在 C++ 中，可通过声明一个类来创建一种新类型。类将一组变量（它们的类型通常不同）和一组相关的函数组合在一起。

一种看待汽车的方式是，将其视为车轮、车门、座位、车窗等的集合；另一种方式是考虑其功能，它可以行驶、加速、减速、刹车、停车等。类让您能够将这些部件和功能封装到一个集合中，这种集合被称为对象。

对程序员来说，将有关汽车的一切封装到类中有很多优点。所有一切都在一个地方，这使引用、复制和调用处理数据的函数很容易。同样，类的客户（使用类的程序代码）可以直接使用对象，而不必考虑它包含什么以及它是如何工作的。

类可以由各种类型的变量组成，还可以包含其他类对象。成员变量也称为数据成员，它们是类中

的变量。类 Car 可能有表示座位数、收音机类型、轮胎等的成员变量。

成员变量也叫数据成员，它们是类中的变量。成员变量是类的组成部分，就像车轮和发动机是汽车的一部分一样。

类还可以包含函数，它们被称为成员函数或方法。成员函数和成员变量一样，也是类的组成部分，它们决定了类的功能。

类中的成员函数通常操纵成员变量。例如，类 Car 可能包含方法 Start() 和 Brake()。类 Cat 可能包含表示年龄和重量的数据成员，其方法可能包括 Sleep()、Meow() 和 ChaseMice()。

### 10.3.1 声明类

类声明将有关类的信息告诉编译器。要声明类，可使用关键字 class，后跟类名、左大括号、数据成员列表和方法，然后是右大括号和分号。下面是 Cat 类的声明：

```
class Cat
{
    unsigned int  itsAge;
    unsigned int  itsWeight;
    void Meow();
};
```

上述声明并没有为 Cat 分配内存，它只是告诉编译器 Cat 是什么：它包含哪些数据 (itsAge 和 itsWeight)，有何功能 (Meow())。虽然没有分配内存，但声明确实让编译器知道 Cat 有多大，即编译器必须为您创建的每个 Cat 对象预留多少内存。在这个例子中，如果 int 变量占 4 字节，则 Cat 的大小为 8 字节：itsAge 占 4 字节，itsWeight 占 4 字节。Meow() 只占用存储有关它所处位置的信息所需的空空间，这是一个函数指针，在 32 位平台上，占用 4 字节。

### 10.3.2 有关命名规则的说明

作为程序员，必须给所有成员变量、成员函数和类命名。正如第 3 章指出的，这些名称应易于理解且有意义。Cat、Rectangle、Employee 都是很好的类名。而 Meow()、ChaseMice() 和 StopEngine 都是很好的函数名，因为它们指出了函数的功能。很多程序员在成员变量名中使用前缀 its，如 itsAge、itsWeight、itsSpeed。这有助于将成员变量和非成员变量区分开来。

有些程序员使用其他的前缀，如 myAge、myWeight、mySpeed。还有一些人使用字母 m，或再加上下划线，如 mAge 或 m\_age、mWeight 或 m\_weight、mSpeed 或 m\_speed。

有些程序员喜欢在类名称前加一个特殊字符，如 cCat 和 cPerson，而另外一些人喜欢全部大写或小写。本书采用首字母大写的方式命名所有类，如 Cat 和 Person。

同样，很多程序员采用首字母大写的方式表示函数，用首字母小写的方式表示变量。单词之间通常用下划线分隔，如 Chase\_Mice，或者将每个单词的首字母大写，如 ChaseMice 和 DrawCircle。

重要的是，应选择一种风格，并在整个程序中始终使用这种风格。随着时间的推移，您的风格会发展到涵盖命名规则、缩进、大括号对齐方式和注释风格。

#### 注意

软件开发公司通常在风格方面有内部标准。这可确保所有开发人员都轻松地读懂其他开发人员编写的代码。不幸的是，这种趋势延伸到了开发操作系统和可重用类库的公司，这通常意味着 C++ 程序必须处理多种不同的命名规则。

#### 警告

正如以前指出的，C++ 是区分大小写的，因此所有的类、函数和变量名应遵循相同的模式，这样就无需考虑拼写：是 Rectangle、rectangle 还是 RECTANGLE。

### 10.3.3 定义对象

声明一个类后，便可以将其用作新类型来声明这种类型的变量。声明新类型对象的方式与声明整型变量相同：

```
unsigned int GrossWeight;    // define an unsigned integer
Cat Frisky;                 // define a Cat
```

上述代码定义了一个名为 `GrossWeight` 的变量，其类型为 `unsigned int`，还定义了一个 `Cat` 类对象 `Frisky`。

### 10.3.4 类与对象

您不会将猫的定义视为宠物，而是将具体的猫作为宠物。您能够区分猫的概念与在起居室到处跑的猫。同样，C++也作为猫概念的 `Cat` 类和每个 `Cat` 对象进行区分。因此，`Frisky` 是一个类型为 `Cat` 的对象，就像 `GrossWeight` 是一个类型为 `unsigned int` 的变量一样。

对象是类的实例。

## 10.4 访问类成员

定义实际的 `Cat` 对象（如 `Cat Frisky;`）后，便可以使用句点运算符（`.`）来访问该对象的成员。因此，要将 50 赋给 `Frisky` 的成员变量 `Weight`，可以这样编写代码：

```
Frisky.itsWeight = 50;
```

同样，要调用 `Meow()` 函数，可以这样编写代码：

```
Frisky.Meow();
```

使用类方法时，调用了该方法。在这个例子中，对 `Frisky` 调用了方法 `Meow()`。

### 10.4.1 给对象而不是类赋值

在 C++ 中，只能给变量赋值，而不能给类型赋值。例如，绝不要编写这样的代码：

```
int = 5;                // wrong
```

编译器将认为这条语句是错误的，因为不能将 5 赋给整型，而必须定义一个整型变量，然后将 5 赋给该变量。例如：

```
int x;                  // define x to be an int
x = 5;                  // set x's value to 5
```

这段代码的意思是：将 5 赋给类型为 `int` 的变量 `x`。同样，也不能编写这样的代码：

```
Cat.itsAge=5;           // wrong
```

编译器将视该语句为错误，因为不能将 5 赋给 `Cat` 类的年龄部分，而必须先定义一个 `Cat` 对象，然后再将 5 赋给该对象，例如：

```
Cat Frisky;             // just like int x;
Frisky.itsAge = 5;       // just like x = 5;
```

### 10.4.2 类不能有没有声明的功能

请做这样一个实验：走到一个 3 岁小孩面前，给他看一只猫，然后说：“这是 `Frisky`，`Frisky` 会耍把戏。`Frisky`，汪汪”。这个小孩会咯咯地笑着说：“不，您真笨，猫不会像狗那样叫”。

同样，如果编写如下代码：

```
Cat Frisky;           // make a Cat named Frisky
Frisky.Bark()         // tell Frisky to bark
```

编译器会说：不，笨蛋，猫不会汪汪叫（编译器的提示语可能是 “[531]Error: Member function Bark not found in class Cat.”）。编译器之所以知道 Frisky 不能汪汪叫，是因为 Cat 类没有 Bark() 函数。如果没有定义 Meow() 函数，编译器甚至不会让 Frisky 喵喵叫。

应该	不应该
一定要使用关键字 class 来声明类。 使用句点运算符 (.) 来访问类的成员和函数。	不要把声明与定义混为一谈。声明指出类是什么，定义为对象分配内存。 不要将类与对象混为一谈。 不要将值赋给类，而应将其赋给对象的数据成员。

### 10.5 私有和公有

在类声明中还常常使用其他关键字，其中最重要的两个是 public 和 private。  
关键字 public 和 private 用于类的成员：数据成员和成员方法。私有成员只能在类的方法中访问，公有成员可以通过类的任何对象进行访问。这种区分既重要又令人迷惑。默认情况下，所有类成员都是私有的。

为解释得清楚点，来看本章前面的一个例子：

```
class Cat
{
    unsigned int  itsAge;
    unsigned int  itsWeight;
    void Meow();
};
```

在这个声明中，itsAge、itsWeight 和 Meow() 都是私有的，因为类的所有成员默认均为私有。这意味着除非特别说明，否则它们都是私有的。如果您创建一个程序，并在 main() 中编写如下代码：

```
int main()
{
    Cat Boots;
    Boots.itsAge=5;           // error! can't access private data!
    ...
}
```

编译器将认为有错误。通过让这些成员为私有的，您实际上告诉了编译器，我将只在 Cat 类的成员函数中访问 itsAge、itsWeight 和 Meow()。而在上述代码中，您从 Cat 方法的外部访问 Boots 对象的成员变量 itsAge。仅仅由于 Boots 是 Cat 类的一个对象，并不意味着可以访问 Boots 的私有部分。

C++ 新手经常会对此感到迷惑。几乎每天都有人在喊：喂！我将 Boots 定义为一个 Cat，为什么 Boots 不能访问自己的年龄？答案是 Boots 能，但您不能。Boots 在自己的方法中可以访问所有部分：私有部分和公有部分。虽然您创建了一个 Cat，但这并不意味着您可以看到或修改它的私有部分。

要能够访问 Cat 的数据成员，可以将有些成员声明为公有的：

```
class Cat
{
    public:
        unsigned int  itsAge;
        unsigned int  itsWeight;
        void Meow();
};
```

在这个声明中，itsAge、itsWeight 和 Meow() 都为公有，这样前面的 Boots.itsAge=5 将能够通过编译，而不会出现任何问题。



**注意**

关键字 `public` 将被应用于声明中关键字 `private` 之前的所有成员，反之亦然。这让您能够轻松地声明类的私有部分和公有部分。

程序清单 10.1 是包含公有成员变量的类 `Cat` 的声明。

程序清单 10.1 访问一个简单类的公有成员

```
1: // Demonstrates declaration of a class and
2: // definition of an object of the class
3:
4: #include <iostream>
5:
6: class Cat           // declare the Cat class
7: {
8:     public:         // members that follow are public
9:         int itsAge;   // member variable
10:        int itsWeight; // member variable
11: };                // note the semicolon
12:
13: int main()
14: {
15:     Cat Frisky;
16:     Frisky.itsAge = 5; // assign to the member variable
17:     std::cout << "Frisky is a cat who is " ;
18:     std::cout << Frisky.itsAge << " years old.\n";
19:     return 0;
20: }
```

**▼ 输出:**

Frisky is a cat who is 5 years old.

**▼ 分析:**

第 6 行包含关键字 `class`，这告诉编译器，接下来是一个声明。新类的名称位于关键字 `class` 的后面，这里为 `Cat`。

声明体从第 7 行的左大括号开始，到第 11 行的右大括号和分号结束。第 8 行包含关键字 `public` 和一个冒号，这表示接下来所有的成员均为公有，直到遇到 `private` 或到达类声明末尾为止。

第 9 行和第 10 行包含类成员 `itsAge` 和 `itsWeight` 的声明。

从第 13 行开始是程序的 `main()` 函数。第 15 行将 `Frisky` 定义为一个 `Cat` 实例，即 `Cat` 对象。在第 16 行，`Frisky` 的年龄被设置为 5。第 17 行和第 18 行使用成员变量 `itsAge` 打印了一条有关 `Frisky` 的消息。读者应注意在第 16 行和第 18 行是如何访问对象 `Frisky` 的成员的，`itsAge` 是使用对象名（这里为 `Frisky`）、句点和成员名（这里为 `itsAge`）访问的。

**注意**

试着注释掉第 8 行，再重新编译。将收到一条第 16 行有错的消息，因为 `itsAge` 将不再是公有成员。注释掉这行后，`itsAge` 和其他成员将默认为私有的。

## 使数据成员为私有的

一个通用的设计规则是，应让类的数据成员为私有的。当然，您可能会问，如果将所有数据成员都设置为私有的，如何访问有关类的信息。例如，如果 `itsAge` 是私有的，如何能够获得或设置 `Cat` 对象的年龄。

为访问类中的私有数据，必须创建被存取器方法（accessor method）的公有函数。这些方法用于设置和获取私有成员变量，它们是成员函数，可以在程序的其他地方调用它们来获取和设置私有成员变量。

公有存取器方法是类成员函数，用于读取或设置私有类成员变量的值。



为什么要费力地使用这种额外的间接访问呢？在直接使用数据更简单也更容易的情况下，为何要添加这些额外的函数呢。为何要通过存取器函数来工作？

这些问题的答案是，存取器函数让您能够将数据的存储细节与数据的使用细节分开。通过使用存取器函数，以后修改数据的存储方式时，不必重新编写使用这些数据的函数。

如果需要知道 Cat 的年龄的函数直接访问 itsAge，当 Cat 类的作者决定改变该数据的存储方式时，必须重新编写该函数。通过让函数调用 GetAge()，Cat 类可以轻松地返回正确的值，而不管得到年龄的方式如何。调用函数不需要知道年龄是被存储在 unsigned int 变量或 long 变量中，还是在需要时计算得到的。

这种技术使得程序更容易维护。它延长了代码的生命周期，因为设计的变化不会导致程序作废。

另外，存取器函数还可以包含其他逻辑。例如，如果 Cat 的年龄不太可能超过 100 岁或其重量不太可能超过 1000，在这种情况下，可以禁止这些值。存取器函数可以实施这种限制，还可能完成其他任务。

程序清单 10.2 对 Cat 类进行了修改，使之包含私有成员数据和公有存取器函数。注意，这不是可运行的程序。

程序清单 10.2 包含存取器方法的类

---

```

1: // Cat class declaration
2: // Data members are private, public accessor methods
3: // mediate setting and getting the values of the private data
4:
5: class Cat
6: {
7:     public:
8:         // public accessors
9:         unsigned int GetAge();
10:        void SetAge(unsigned int Age);
11:
12:        unsigned int GetWeight();
13:        void SetWeight(unsigned int Weight);
14:
15:        // public member functions
16:        void Meow();
17:
18:        // private member data
19:    private:
20:        unsigned int  itsAge;
21:        unsigned int  itsWeight;
22: };

```

---

#### ▼ 分析：

这个类有 5 个公有方法。第 8 行和第 9 行包含 itsAge 的存取器方法，正如读者看到的，第 8 行有一个获取年龄的方法，第 9 行有一个设置年龄的方法。第 11 行和第 12 行包含 itsWeight 的存取器方法。这些存取器函数设置和返回成员变量的值。

第 15 行声明了公有成员函数 Meow()，它不是存取器函数：不获取或设置成员变量的值，而是为这个类提供另一项服务——打印单词 Meow。

第 19 行和第 20 行声明了成员变量。

要设置 Frisky 的年龄，可以将年龄传递给 SetAge() 方法，如下所示：

```

Cat Frisky;
Frisky.SetAge(5);    // set Frisky's age using the public accessor

```

本章后面将列出 SetAge() 和其他方法的代码。

将方法或数据声明为私有的让编译器能够发现编程错误，避免它们成为 bug。只要愿意支付咨询费，任何程序员都可以获得避开私有性的方法。C++ 的发明者 Stroustrup 说：C++ 的访问控制机理用于防范意外事件，而不是保护欺骗。

关键字 **class**

关键字 `class` 的语法如下:

```
class class_name
{
    // access control keywords here
    // class variables and methods declared here
};
```

关键字 `class` 用于声明新类型。类是类成员数据的集合，类成员数据可以是各种类型的变量，包括其他类。类还包含类函数（方法），即用来操纵类中的数据或为类提供其他服务的函数。

定义类对象的方法与定义变量一样。首先指出类型（类），然后是变量名（对象）。使用句点运算符（.）可访问类的成员和函数。

使用访问控制关键字可将类的某部分声明为私有或公有的。默认的访问控制为私有。访问控制关键字将改变从当前位置到下个关键字或类结尾之间所有成员的访问控制。类声明以右大括号和分号结尾。

示例 1:

```
class Cat
{
    public:
        unsigned int Age;
        unsigned int Weight;
        void Meow();
};

Cat Frisky;
Frisky.Age = 8;
Frisky.Weight = 18;
Frisky.Meow();
```

示例 2:

```
class Car
{
    public:                                // the next five are public

        void Start();
        void Accelerate();
        void Brake();
        void SetYear(int year);
        int GetYear();

    private:                               // the rest is private

        int Year;
        Char Model [255];
};                                          // end of class declaration

Car OldFaithful;                          // make an instance of car
int bought;                               // a local variable of type int
OldFaithful.SetYear(84);                  // assign 84 to the year
bought = OldFaithful.GetYear();           // set bought to 84
OldFaithful.Start();                      // call the start method
```

应该

务必将存取器方法声明为公有。  
必须通过类的成员函数来访问私有成员变量。

不应该

尽可能不要将成员变量声明为公有的。  
不要试图在类的外部使用私有成员变量。

10.6 实现类方法

正如读者看到的，存取器函数提供了到类的私有成员数据的公有接口。每个存取器函数以及声明

的其他类方法都必须有实现。实现被称为函数的定义。

成员函数的定义类似于常规函数：首先指出函数的返回类型，如果函数不返回任何值，则使用 `void`。然后是类名、两个冒号、函数名和参数。程序清单 10.3 给出了简单的 `Cat` 类的完整声明及存取器函数和一个普通类成员函数的实现。

程序清单 10.3 实现一个简单类的方法

```

1: // Demonstrates declaration of a class and
2: // definition of class methods
3: #include <iostream>           // for cout
4:
5: class Cat                     // begin declaration of the class
6: {
7:     public:                   // begin public section
8:         int GetAge();          // accessor function
9:         void SetAge (int age); // accessor function
10:        void Meow();           // general function
11:    private:                   // begin private section
12:        int itsAge;            // member variable
13: };
14:
15: // GetAge, Public accessor function
16: // returns value of itsAge member
17: int Cat::GetAge()
18: {
19:     return itsAge;
20: }
21:
22: // definition of SetAge, public
23: // accessor function
24: // sets itsAge member
25: void Cat::SetAge(int age)
26: {
27:     // set member variable itsAge to
28:     // value passed in by parameter age
29:     itsAge = age;
30: }
31:
32: // definition of Meow method
33: // returns: void
34: // parameters: None
35: // action: Prints "meow" to screen
36: void Cat::Meow()
37: {
38:     std::cout << "Meow.\n";
39: }
40:
41: // create a cat, set its age, have it
42: // meow, tell us its age, then meow again.
43: int main()
44: {
45:     Cat Frisky;
46:     Frisky.SetAge(5);
47:     Frisky.Meow();
48:     std::cout << "Frisky is a cat who is " ;
49:     std::cout << Frisky.GetAge() << " years old.\n";
50:     Frisky.Meow();
51:     return 0;
52: }

```

#### ▼ 输出:

```

Meow.
Frisky is a cat who is 5 years old.
Meow.

```

#### ▼ 分析:

第 5~13 行是 `Cat` 类的定义。第 7 行的关键字 `public` 告诉编译器，接下来是一组公有成员。第 8

行声明了公有存取器方法 `GetAge()`。`GetAge()`让您能够访问第 12 行定义的私有成员变量 `itsAge`。第 9 行声明了公有存取器函数 `SetAge()`。`SetAge()`接受一个 `int` 参数并将 `itsAge` 设置为该参数的值。

第 10 行声明了类方法 `Meow()`。`Meow()`不是存取器函数，而是一个将单词 `Meow` 打印到屏幕上的普通方法。

第 11 行开始为私有部分，其中只包含第 12 行的私有成员变量 `itsAge` 的声明。第 13 行用右大括号和分号结束类声明。

第 17~20 行包含成员函数 `GetAge()` 的定义。这个函数没有参数，但返回一个 `int` 值。从第 17 行可知，类方法包括类名、两个冒号和函数名。这种语法告诉编译器，这里定义的函数 `GetAge()` 是在 `Cat` 类中声明的那个。除函数头外，`GetAge()` 函数的创建方法与任何其他函数相同。

`GetAge()` 函数只有一行，它返回 `itsAge` 的值。注意，由于 `itsAge` 是 `Cat` 类私有的，因此在 `main()` 函数中不能直接访问它，但 `main()` 可以访问公有方法 `GetAge()`。

由于 `GetAge()` 是 `Cat` 类的一个成员函数，它可以访问变量 `itsAge`。这使得 `GetAge()` 能够将 `itsAge` 的值返回给 `main()`。

第 25 行是成员函数 `SetAge()` 的定义，它接受一个 `int` 参数 (`age`)，且不返回任何值 (用 `void` 表示)。`SetAge()` 接受参数 `age` 的值，将其赋给 `itsAge` (第 29 行)。由于 `SetAge()` 函数也是 `Cat` 类的成员，因此可以直接访问私有成员变量 `itsAge`。

从第 36 行开始是 `Cat` 类的 `Meow()` 方法的定义 (实现)。这个函数只有一行，它将单词 `Meow` 打印到屏幕上并换行。还记得吗，字符 `\n` 表示在屏幕上换行。`Meow()` 与存取器函数类似，也以返回类型打头，然后是类名、函数名和参数 (这个函数没有)。

从第 43 行开始是程序体，也包含函数 `main()`。在第 45 行，`main()` 函数声明了一个名为 `Frisky` 的类型为 `Cat` 的对象；也可以这样说，`main()` 声明一个名为 `Frisky` 的 `Cat`。

在第 46 行，通过存取器方法 `SetAge()` 将 5 赋给成员变量 `itsAge`。注意，该方法是使用对象名 (`Frisky`)、句点运算符 (`.`) 和方法名 (`SetAge()`) 调用的。也可以以类似的方式调用类中的其他方法。

#### 注意

术语成员函数与方法的含义相同。

第 47 行调用成员函数 `Meow()`，第 49 行使用 `GetAge()` 方法打印一条消息。第 50 行再次调用 `Meow()`。虽然这些方法都是类 (`Cat`) 的一部分，且需要通过对象 (`Frisky`) 来调用，但其运行方式与常规函数相同。

## 10.7 添加构造函数和析构函数

定义整型变量的方法有两种。一种是先定义该变量，然后在程序中为其赋值。例如：

```
int Weight;           // define a variable
...                   // other code here
Weight = 7;           // assign it a value
```

另一种方法是定义该变量并立即对其初始化。例如：

```
int Weight = 7;       // define and initialize to 7
```

初始化将变量的定义和赋初值合而为一。以后可以随意修改这个值。初始化确保变量总是有一个有意义的值。

如何初始化类的成员数据呢？可以使用一个特殊的成员函数：构造函数。构造函数可以根据需要接受参数，但它不能有返回值——连 `void` 都不行。构造函数是一个与类同名的类方法。

声明构造函数后，还应声明析构函数。构造函数创建并初始化类对象，而析构函数在对象被销毁后完成清理工作并释放 (在构造函数或对象的生命周期中) 分配的资源或内存。析构函数总是与类同名，但在前面加上了一个 `~`。析构函数没有参数，也没有返回值。如果要为 `Cat` 类声明一个析构函数，



声明与下面类似：

```
~ Cat();
```

### 10.7.1 默认构造函数和析构函数

有很多种类型的构造函数，有些接受参数，有些不接受。没有参数的构造函数被称为默认构造函数。只有一个析构函数参数，和默认构造函数一样，它也不接受任何参数。

如果设有创建构造函数或析构函数，编译器将为您提供一个。编译器提供的构造函数是默认构造函数。

编译器创建的默认构造函数和析构函数不仅没有参数，而且好像不执行任何操作！如果希望它们执行一些操作，必须创建自己的默认构造函数或析构函数。

### 10.7.2 使用默认构造函数

什么也不做的构造函数有什么好处呢？在某种程度上说，这是一个格式的问题。所有对象都必须被构造和析构，在构造和析构过程中，将调用这些不执行任何操作的函数。

要在声明对象时不用传递任何参数，如：

```
Cat Rags;           // Rags gets no parameters
```

必须有一个下面这样的构造函数：

```
Cat();
```

创建类对象时，将调用构造函数。如果 Cat 构造函数接受两个参数，应这样创建 Cat 对象：

```
Cat Frisky (5,7);
```

在这个例子中，第一个参数可能是年龄，第二个参数可能是重量。

如果构造函数接受一个参数，应这样编写代码：

```
Cat Frisky (3);
```

如果构造函数根本不接受参数（也就是说，它是默认构造函数），应去掉括号，这样编写代码：

```
Cat Frisky;
```

通常调用函数时，即使函数不接受任何参数也应提供括号，但上面是一种例外情况。这就可以编写如下代码的原因：

```
Cat Frisky;
```

这行代码被解释为调用默认构造函数。没有提供任何参数，因此将括号删除。

注意，并非一定要使用编译器提供的默认构造函数，可以编写自己的默认构造函数：不接受任何参数的构造函数。也可以给自己的默认构造函数提供函数体，在其中对对象进行初始化。出于格式方面的考虑，建议至少定义一个构造函数，将成员变量指定合适的默认值，以确保对象总是能正确地运行。

同样出于格式方面的考虑，如果声明了一个构造函数，一定要声明一个析构函数，哪怕该析构函数什么也不做。虽然默认析构函数可以正常工作，但声明自己的析构函数并没有坏处。这将让代码更清晰。

程序清单 10.4 重写了 Cat 类，使用一个非默认构造函数来初始化 Cat 对象，将其年龄设置为您提供的初始年龄。该程序清单还演示了在什么地方调用析构函数。

#### 程序清单 10.4 使用构造函数和析构函数

```
1: // Demonstrates declaration of constructors and
2: // destructor for the Cat class
3: // Programmer created default constructor
4: #include <iostream>           // for cout
5:
6: class Cat                    // begin declaration of the class
7: {
8:     public:                  // begin public section
```



```
9:     Cat(int initialAge); // constructor
10:     ~Cat(); // destructor
11:     int GetAge(); // accessor function
12:     void SetAge(int age); // accessor function
13:     void Meow();
14:     private: // begin private section
15:         int itsAge; // member variable
16: };
17:
18: // constructor of Cat,
19: Cat::Cat(int initialAge)
20: {
21:     itsAge = initialAge;
22: }
23:
24: Cat::~~Cat() // destructor, takes no action
25: {
26: }
27:
28: // GetAge, Public accessor function
29: // returns value of itsAge member
30: int Cat::GetAge()
31: {
32:     return itsAge;
33: }
34:
35: // Definition of SetAge, public
36: // accessor function
37: void Cat::SetAge(int age)
38: {
39:     // set member variable itsAge to
40:     // value passed in by parameter age
41:     itsAge = age;
42: }
43:
44: // definition of Meow method
45: // returns: void
46: // parameters: None
47: // action: Prints "meow" to screen
48: void Cat::Meow()
49: {
50:     std::cout << "Meow.\n";
51: }
52:
53: // create a cat, set its age, have it
54: // meow, tell us its age, then meow again.
55: int main()
56: {
57:     Cat Frisky(5);
58:     Frisky.Meow();
59:     std::cout << "Frisky is a cat who is " ;
60:     std::cout << Frisky.GetAge() << " years old.\n";
61:     Frisky.Meow();
62:     Frisky.SetAge(7);
63:     std::cout << "Now Frisky is " ;
64:     std::cout << Frisky.GetAge() << " years old.\n";
65:     return 0;
66: }
```

---

#### ▼ 输出:

```
Meow.
Frisky is a cat who is 5 years old.
Meow.
Now Frisky is 7 years old.
```

---

#### ▼ 分析:

除第 9 行添加了一个接受一个 int 参数的构造函数外, 程序清单 10.4 与程序清单 10.3 极其相似。第 10 行声明了析构函数, 它没有参数。析构函数不能有参数, 构造函数和析构函数都没有返回值——连 void 也没有。

第 19~22 行是构造函数的实现，它与存取器函数 `SetAge()` 的实现类似。正如读者看到的，构造函数名前加上了类名。前面指出过，这指出方法（这里为 `Cat()`）是 `Cat` 类的一部分。这是一个构造函数，因此没有返回值——连 `void` 也没有。然而，该构造函数接受一个初始值，在第 21 行，这个初始值被赋给了数据成员 `itsAge`。

第 24~26 行是析构函数 `~Cat()` 的实现。该函数不执行任何操作，但如果在类声明中声明了析构函数，必须包含其定义。与构造函数和其他方法一样，在析构函数名前也加上了类名。与构造函数一样（但不同于其他方法），析构函数没有返回类型，也没有参数。这是一个有关析构函数的标准。

第 57 行创建了一个 `Cat` 对象 `Frisky`，并将 5 传递给构造函数。从第 60 行可知，不需要调用 `SetAge()` 函数，因为创建 `Frisky` 时已经将其成员变量 `itsAge` 设置为 5。在第 62 行，将 `Frisky` 的成员变量 `itsAge` 重新设置为 7。第 64 行打印新的值。

应该	不应该
一定要使用构造函数来初始化对象。 添加构造函数后，一定要添加一个析构函数。	不要让构造函数和析构函数有返回值。 不要让析构函数接受任何参数。

## 10.8 const 成员函数

在本书前面，读者使用关键字 `const` 来声明不能修改的变量。也可将关键字 `const` 用于类中的成员函数。如果将类方法声明为 `const`，必须保证该方法不会修改任何类成员的值。

要将类方法声明为 `const`，可在方法声明中将所有参数括起的括号和分号之间放置关键字 `const`，例如：  
`void SomeFunction() const;`

这声明了一个名为 `SomeFunction()` 的 `const` 成员方法，它不接受任何参数，返回类型为 `void`。由于它被声明为 `const` 的，因此不会修改其所属类的任何数据成员。

通常使用修饰符 `const` 将只读取值的存取器函数声明为 `const` 函数。前面的 `Cat` 类有两个存取器函数：  
`void SetAge(int anAge);`  
`int GetAge();`

函数 `SetAge()` 不能是 `const` 的，因为它修改成员变量 `itsAge` 的值；而 `GetAge()` 应该是 `const` 的，因为它不修改类的任何成员。

`GetAge()` 只返回成员变量 `itsAge` 的当前值。因此这些函数的声明应该写成这样：

```
void SetAge(int anAge);  
int GetAge() const;
```

如果将一个函数声明为 `const` 的，而该函数的实现通过修改某个成员变量的值而修改了对象，编译器将视其为错误。例如，如果将前面的 `GetAge()` 声明为 `const` 的，而记录询问 `Cat` 年龄的次数，将产生编译错误。这是因为调用该方法将修改 `Cat` 对象。

一种良好的编程习惯是，尽可能将方法声明为 `const` 的。这样让编译器捕获错误，而不致于成为等到程序运行时才出现的 `bug`。

### 为什么使用编译器来捕获错误？

编写 100% 正确的代码当然很好，但很少有程序员能做到这一点。然而，很多程序员开发了一个系统，通过在编程初期找出并修复错误，来最大限度地减少 `bug`。

虽然编译器错误令人头痛，对程序员的生存也是一种挑战，但与其他错误相比，这种错误要好得多。对于弱类型语言，当您违反合同时，编译器不会发出错误信号，但是当您的老板正在您旁边观看您运行程序时，程序可能崩溃。更糟糕的是，在捕获错误方面，测试的帮助相对较小，因为对于实际的程序而言，

能够通过测试的路径实在太多了。

编译错误（编译时出现的错误）比运行错误（运行程序时出现的错误）要好得多，因为编译错误被查出的可能性很大。即使运行程序多次，也可能没有经过每个可能的代码路径。因此，运行错误可能潜伏很长时间，而编译错误在每次编译时都会出现，因此更容易发现和修复。高质量编程的目标是，确保代码没有运行错误。为实现这种目标，一种经过实践检验的方法是，使用编译器在开发初期捕获错误。

## 10.9 将类声明和方法定义放在什么地方

为类声明的每个函数都必须有定义，这种定义称为函数实现。与其他函数一样，类方法的定义也由函数头和函数体组成。

定义必须位于编译器能够找到的文件中，大多数 C++ 编译器希望这种文件的扩展名为 .c 或 .cpp。本书使用 .cpp，但请了解您的编译器要求使用哪种扩展名。

### 注意

很多编译器假定扩展名为 .c 的文件为 C 程序，而 C++ 程序使用扩展名 .cpp。您可以使用任何扩展名，但使用 .cpp 可最大限度地避免混淆。

也可以将声明放在实现文件中，但这不是一种好的编程习惯。大多数程序员采用的约定是，将声明放在头文件中，该头文件的名称与实现文件相同，但扩展名为 .h、.hp 或 .hpp。本书将 .h 用作头文件的扩展名，但请了解您的编译器要求使用何种扩展名。

例如，可以将 Cat 类的声明放在一个名为 Cat.h 的文件中，而将类方法的定义放在一个名为 Cat.cpp 的文件中。然后，在 Cat.cpp 的开头加入如下代码，将头文件同 .cpp 文件关联起来：

```
#include "Cat.h"
```

这段代码告诉编译器，将 Cat.h 读入文件中，就好像在这里直接输入了头文件的内容一样。注意，有些编译器要求 #include 语句和文件系统中的文件名大小写一致。

如果只想将 .hpp 文件的内容读入 .cpp 文件中，又何必将它们分成两个文件呢？在大多数情况下，类的客户并不关心实现细节。只要阅读头文件，就可以知道需要知道的所有信息；他们可以忽略实现文件。另外，很可能需要在 .cpp 文件中包含同一个 .hpp 文件。

### 注意

类声明告诉编译器：类是什么、它存储什么样的数据、有什么样的函数。之所以将类声明称为接口，是因为它告诉用户如何与类打交道。接口通常存储在扩展名为 .hpp 的头文件中。函数定义告诉编译器，函数是如何工作的。函数定义被称为类方法的实现，存储在一个 .cpp 文件中。类的实现细节只有类的作者关心；类的客户（即使用类的那部分程序）不需要知道，也不关心函数是如何实现的。

## 10.10 内联实现

就像可以请求编译器将常规函数作为的内联一样，也可以将类方法作为内联的，为此只需在返回类型前加上关键字 inline。例如，GetWeight() 函数的内联实现如下：

```
inline int Cat::GetWeight()
{
    return itsWeight;    // return the Weight data member
}
```

也可以将函数的定义放到类声明中，这样，函数将自动成为内联的。例如：

```
class Cat
{
public:
    int GetWeight() { return itsWeight; } // inline
    void SetWeight(int aWeight);
};
```

请注意 `GetWeight()` 定义的语法。内联函数的函数体紧跟在类方法声明之后，圆括号后面没有分号。与其他函数一样，该定义以左大括号开始，以右大括号结束。和往常一样，空白无关紧要，可以这样书写上述声明：

```
class Cat
{
public:
    int GetWeight() const
    {
        return itsWeight;
    } // inline
    void SetWeight(int aWeight);
};
```

程序清单 10.5 和 10.6 重新编写了 `Cat` 类，将类声明放在文件 `Cat.h` 中，将函数的实现放在文件 `Cat.cpp` 中。程序清单 10.5 还将存取器函数和 `Meow()` 函数改为内联的。

程序清单 10.5 位于 `Cat.h` 中的 `Cat` 类声明

---

```
1: #include <iostream>
2: class Cat
3: {
4:     public:
5:         Cat (int initialAge);
6:         ~Cat();
7:         int GetAge() const { return itsAge;} // inline!
8:         void SetAge (int age) { itsAge = age;} // inline!
9:         void Meow() const { std::cout << "Meow.\n";} // inline!
10:     private:
11:         int itsAge;
12: };
```

---

程序清单 10.6 位于 `Cat.cpp` 中的 `Cat` 类实现

---

```
1: // Demonstrates inline functions
2: // and inclusion of header files
3: // be sure to include the header files!
4: #include "Cat.h"
5:
6:
7: Cat::Cat(int initialAge) //constructor
8: {
9:     itsAge = initialAge;
10: }
11:
12: Cat::~Cat() //destructor, takes no action
13: {
14: }
15:
16: // Create a cat, set its age, have it .
17: // meow, tell us its age, then meow again.
18: int main()
19: {
20:     Cat Frisky(5);
21:     Frisky.Meow();
22:     std::cout << "Frisky is a cat who is " ;
23:     std::cout << Frisky.GetAge() << " years old.\n";
24:     Frisky.Meow();
25:     Frisky.SetAge(7);
26:     std::cout << "Now Frisky is " ;
27:     std::cout << Frisky.GetAge() << " years old.\n";
28:     return 0;
29: }
```

---

#### ▼ 输出:

---

```
Meow.
Frisky is a cat who is 5 years old.
Meow.
Now Frisky is 7 years old.
```



### ▼ 分析:

程序清单 10.5 和 10.6 中的代码与程序清单 10.4 中的代码相似, 只是将声明放到了 Cat.h (程序清单 10.5) 中, 同时在声明中将 3 个方法声明为内联的。

Cat.h 的第 6 行声明了 GetAge(), 并提供了其内联实现。第 7 行和第 8 行给出了另外两个内联函数。但这些内联函数功能与前一个例子中相同。

Cat.cpp (程序清单 10.6) 的第 4 行为#include "Cat.h", 它导入 Cat.h 的内容。通过包含 Cat.h, 告诉预编译器将 Cat.h 读入到文件中, 就像在这里 (第 5 行) 输入了该文件的内容一样。

这种技术让您能够将声明和实现放在不同的文件中, 同时让编译器需要时可以使用声明。这是 C++ 编程中一项非常常见的技术。通常, 将类声明放在一个 .hpp 文件中, 然后使用#include 语句将其包含到相关的 .cpp 文件中。

第 18~29 行与程序清单 10.4 中的 main() 函数相同, 用于证明将这些函数声明为内联的并不会改变其功能。

## 10.11 将其他类用作成员数据的类

一种常见的创建复杂类的方法是, 首先声明较简单的类, 然后将其包含到较复杂类的声明中。例如, 您可能声明车轮类、发动机类、离合器类等, 然后将它们组合成汽车类。这声明一种 has-a (有一个) 关系。汽车有发动机、车轮、离合器等。

来看另一个例子。矩形由线段组成, 线段由两点确定, 而点由坐标 x、y 确定。程序清单 10.7 给出了 Rectangle 类的完整声明, 该声明可能存储在文件 Rectangle.h 中。由于矩形被定义为连接 4 个点的 4 条线段, 而每个点对应图上一个坐标, 因此先声明一个 Point 类来存储点的 x、y 坐标。程序清单 10.8 给出了这两个类的实现。

程序清单 10.7 声明一个完整的类

```
1: // Begin Rectangle.h
2: #include <iostream>
3: class Point    // holds x,y coordinates
4: {
5:     // no constructor, use default
6:     public:
7:         void SetX(int x) { itsX = x; }
8:         void SetY(int y) { itsY = y; }
9:         int GetX()const { return itsX;}
10:        int GetY()const { return itsY;}
11:     private:
12:         int itsX;
13:         int itsY;
14: };    // end of Point class declaration
15:
16:
17: class Rectangle
18: {
19:     public:
20:         Rectangle (int top, int left, int bottom, int right);
21:         ~Rectangle () {}
22:
23:         int GetTop() const { return itsTop; }
24:         int GetLeft() const { return itsLeft; }
25:         int GetBottom() const { return itsBottom; }
26:         int GetRight() const { return itsRight; }
27:
28:         Point GetUpperLeft() const { return itsUpperLeft; }
29:         Point GetLowerLeft() const { return itsLowerLeft; }
30:         Point GetUpperRight() const { return itsUpperRight; }
31:         Point GetLowerRight() const { return itsLowerRight; }
```



```

32:
33:     void SetUpperLeft(Point Location) {itsUpperLeft = Location;}
34:     void SetLowerLeft(Point Location) {itsLowerLeft = Location;}
35:     void SetUpperRight(Point Location) {itsUpperRight = Location;}
36:     void SetLowerRight(Point Location) {itsLowerRight = Location;}
37:
38:     void SetTop(int top) { itsTop = top; }
39:     void SetLeft (int left) { itsLeft = left; }
40:     void SetBottom (int bottom) { itsBottom = bottom; }
41:     void SetRight (int right) { itsRight = right; }
42:
43:     int GetArea() const;
44:
45: private:
46:     Point itsUpperLeft;
47:     Point itsUpperRight;
48:     Point itsLowerLeft;
49:     Point itsLowerRight;
50:     int itsTop;
51:     int itsLeft;
52:     int itsBottom;
53:     int itsRight;
54: };
55: // end Rectangle.h

```

---

#### 程序清单 10.8 Rect.cpp

---

```

1: // Begin Rect.cpp
2: #include "Rectangle.h"
3: Rectangle::Rectangle(int top, int left, int bottom, int right)
4: {
5:     itsTop = top;
6:     itsLeft = left;
7:     itsBottom = bottom;
8:     itsRight = right;
9:
10:    itsUpperLeft.SetX(left);
11:    itsUpperLeft.SetY(top);
12:
13:    itsUpperRight.SetX(right);
14:    itsUpperRight.SetY(top);
15:
16:    itsLowerLeft.SetX(left);
17:    itsLowerLeft.SetY(bottom);
18:
19:    itsLowerRight.SetX(right);
20:    itsLowerRight.SetY(bottom);
21: }
22:
23:
24: // compute area of the rectangle by finding sides,
25: // establish width and height and then multiply
26: int Rectangle::GetArea() const
27: {
28:     int Width = itsRight - itsLeft;
29:     int Height = itsTop - itsBottom;
30:     return (Width * Height);
31: }
32:
33: int main()
34: {
35:     //initialize a local Rectangle variable
36:     Rectangle MyRectangle (100, 20, 50, 80 );
37:
38:     int Area = MyRectangle.GetArea();
39:
40:     std::cout << "Area: " << Area << "\n";
41:     std::cout << "Upper Left X Coordinate: ";
42:     std::cout << MyRectangle.GetUpperLeft().GetX();
43:     return 0;
44: }

```

---

**▼ 输出:**

```
Area: 3000  
Upper Left X Coordinate: 20
```

**▼ 分析:**

在文件 `Rectangle.h` (程序清单 10.7) 中, 第 3~14 行声明了 `Point` 类, 用来存储 `x`、`y` 坐标。这个程序对 `Points` 用得不多, 但其他绘图方法需要 `Point`。

**注意**

如果声明一个名为 `Rectangle` 的类, 有些编译器会报告错误。这通常是由于有一个名为 `Rectangle` 的内部类。如果出现这种问题, 只需将类名改为 `myRectangle` 即可。

在 `Point` 类的声明中, 第 12~13 行声明了两个成员变量 (`itsX` 和 `itsY`)。这两个变量用于存储坐标值。`x` 坐标增大时, 点向右移动; `y` 坐标增大时, 点向上移动。有些图形使用不同的坐标系, 例如, 在有些 Windows 程序中, 在窗口中向下移动时 `y` 坐标增加。

`Point` 类使用第 7~10 行声明的内联存取器函数来获取和设置 `x` 和 `y` 的值。`Point` 类使用默认构造函数和析构函数, 因此必须显式地设置点的坐标。

从第 17 行开始是 `Rectangle` 类的声明。`Rectangle` 由表示顶角的 4 个点组成。

第 20 行是 `Rectangle` 的构造函数, 它接受 4 个 `int` 参数, 分别是 `top`、`left`、`bottom` 和 `right`。在程序清单 10.8 中, 这 4 个参数的值被复制到 4 个成员变量中, 然后创建 4 个 `Points`。

除几个普通的存取器函数外, `Rectangle` 还有一个 `GetArea()` 函数, 它是在第 43 行声明的。在程序清单 10.8 的第 28 行和第 29 行, 该函数计算矩形的面积, 而不是将面积存储在一个变量中。为此, 它计算矩形的长和宽, 然后将它们相乘。

要得到矩形左上角的 `x` 坐标, 需要访问点 `UpperLeft`, 请求它提供 `x` 的值。由于 `GetUpperLeft()` 是 `Rectangle` 的一个方法, 因此可以直接访问 `Rectangle` 的私有数据, 包括 `itsUpperLeft`。由于 `itsUpperLeft` 是一个 `Point`, 而 `Point` 的 `itsX` 值是私有的, 因此 `GetUpperLeft()` 不能直接访问 `itsX`, 而必须使用公有存取器函数 `GetX()` 来获取这个值。

实际程序的主体从程序清单 10.8 的第 33 行开始。在第 36 行之前, 还没有分配任何内存, 因此实际上什么也没有发生。到目前为止, 只是告诉了编译器如何创建点和矩形。

在第 36 行, 通过给 `Top`、`Left`、`Bottom` 和 `Right` 传递值, 创建了一个 `Rectangle` 对象。

第 38 行创建了一个 `int` 类型的局部变量 `Area`。该变量用于存储刚才创建的矩形的面积。该变量被初始化为 `Rectangle` 的 `GetArea()` 函数的返回值。`Rectangle` 类的客户不用了解 `GetArea()` 的实现就能够创建一个 `Rectangle` 对象并获取面积。

`Rectangle.h` 如程序清单 10.7 所示。只需阅读这个头文件 (它包含了 `Rectangle` 类的声明), 程序员便知道 `GetArea()` 返回一个 `int`。`Rectangle` 类的用户不关心 `GetArea()` 是如何完成其工作的。实际上, `Rectangle` 的作者可以修改 `GetArea()` 函数, 而不会影响使用 `Rectangle` 类的程序, 只要该函数仍返回一个 `int`。

程序清单 10.8 中的第 42 行看起来有些奇怪, 但只要考虑发生的情况, 便会很清晰。这行代码获取矩形左上角的 `x` 坐标, 它调用了 `Rectangle` 类的 `GetUpperLeft()` 方法, 该方法返回一个 `Point`, 而您想通过该 `Point` 获取其 `x` 坐标。您知道在 `Point` 类中, 获取 `x` 坐标的存取器函数为 `GetX()`。因此, 为获得矩形左上角的 `x` 坐标, 对 `MyRectangle` 对象调用存取器函数 `GetUpperLeft()`, 再对返回值调用存取器函数 `GetX()`, 如第 42 行所示:

```
MyRectangle.GetUpperLeft().GetX();
```

## FAQ

声明与定义的区别是什么？

答：声明只引入一个名称而不为其分配内存，而定义分配内存。

存在一些例外情况，在这些情况下，所有声明也是定义。其中最重要的例外是，全局函数的声明（原型）和类的声明（通常是在头文件中）。

## 10.12 探索结构

与关键字 `class` 极为相似的一个关键字是 `struct`，它用来声明结构。在 C++ 中，结构与类相同，只是其成员默认为公有的，且默认采用公有继承。可以像声明类一样声明结构，并给它声明数据成员和函数。事实上，如果遵循显式地声明类的公有和私有部分这种良好的编程习惯，那么结构和类的声明方法没有任何不同。

请将程序清单 10.7 做如下修改：

- 在第 3 行，将 `class Point` 改为 `struct Point`，
- 在第 17 行，将 `class Rectangle` 改为 `struct Rectangle`。

现在再次运行程序，并将新输出与原来的输出进行比较，您将发现没有任何变化。

读者可能会问，为什么这两个关键字的功能相同呢？这是一个历史遗留问题。C++ 是作为 C 语言的扩展开发的。C 语言有结构，虽然 C 语言中的结构没有类方法。C++ 创始人 Bjarne Stroustrup 对结构进行了扩展，为表明这是一种新的扩展功能，将名称改为类，同时修改了成员的默认可见性。这也使得可以在 C++ 程序中使用庞大的 C 函数库。

## 应该

将类声明放在 .h（头）文件中，将成员函数的实现放在 .cpp 文件中。

尽可能使用 `const`。

## 不应该

理解类之前不要继续学习后面的内容。

## 10.13 总结

本章介绍了如何使用类创建新的数据类型；还介绍了如何定义这些新类型的变量——对象。

类可以有数据成员，数据成员是各种类型（包括其他类）的变量。类还可以包含成员函数，也叫方法。可以使用成员函数来操纵成员数据和提供其他服务。

类成员（包括数据和函数）可以是私有或公有的。程序的任何部分均可访问公有成员。私有成员只能由类的成员函数访问。类成员默认为私有的。

一种良好的编程习惯是，将类的接口（声明）放在头文件中（头文件的扩展名通常为 .h），然后在代码文件（.cpp）中使用 `include` 语句来使用它。类方法的实现通常存放在扩展名为 .cpp 的文件中。

类构造函数可用于初始化对象的数据成员。类析构函数在对象被销毁时执行，它通常用于释放由类方法分配的内存和其他资源。

## 10.14 问与答

问：如何确定类对象的大小？

答：类对象在内存中的大小由类成员变量的大小的总和决定。类方法只占用少量的内存，这些内

存用于存储有关方法位置的信息（指针）。

有些编译器在内存中将变量与某种边界对齐，因此两字节的变量实际占用的内存可能不止两字节。请查看编译器手册，看看是否如此，但就现在而言，读者不用考虑这些细节。

问：如果声明一个包含私有成员 `itsAge` 的 `Cat` 类，然后再定义两个 `Cat` 对象 `Frisky` 和 `Boots`，`Boots` 能访问 `Frisky` 的 `itsAge` 成员变量吗？

答：不能，同一个类的不同实例不能彼此访问对方的非公有成员。换句话说，如果 `Frisky` 和 `Boots` 都是 `Cat` 的实例，那么 `Frisky` 的成员函数可以访问 `Frisky` 的数据，但不能访问 `Boots` 的数据。

问：为什么不应将所有成员数据都声明为公有的？

答：将数据成员声明为私有的，让类的客户能够使用数据而不用关心数据如何存储或计算。例如，假设 `Cat` 类有一个 `GetAge()` 方法，`Cat` 类的客户可以询问猫的年龄，而不用知道或关心猫是将其年龄保存在一个成员变量中还是动态计算得到的。这意味着编写 `Cat` 类的程序员可以在以后修改 `Cat` 类的设计，而不会要求所有的 `Cat` 用户都修改其程序。

问：既然在 `const` 函数对类进行修改会导致编译错误，为什么不能省掉 `const` 以避免错误呢？

答：如果成员函数在逻辑上不应修改类，则通过使用关键字 `const`，可让编译器帮助发现一些错误。例如，`GetAge()` 函数可能没有理由去修改 `Cat` 类，但实现中可能有下面这行代码：

```
if (itsAge == 100) cout << "Hey! You're 100 years old\n";
```

如果将 `GetAge()` 声明为 `const` 的，将导致这段代码被视为错误的。由于您的本意是判断 `itsAge` 是否等于 100，但不小心将 100 赋给了 `itsAge`。由于这种赋值修改了类，而声明中又指出该方法不会修改类，因此编译器能够发现这种错误。

这种错误仅仅通过浏览代码很难发现。人们常常只能看到预期的东西。更重要的是，程序可能显得一切正常，但 `itsAge` 却被设置为一个错误的值，这迟早会导致问题。

问：在 C++ 程序中有什么理由使用结构吗？

答：很多 C++ 程序员使用 `struct` 来声明无任何函数的类。这是沿袭旧的 C 结构的做法。在旧的 C 结构中不能有函数。坦率地讲，这是一种令人迷惑的不良编程习惯。今天无方法结构明天可能需要方法。如果那样的话，要么将其改为类，要么打破您的规则，在结构中添加方法。仅当要调用需要特定结构的老式 C 函数时，才应使用结构。

问：有些从事面向对象编程的人使用术语实例化，这是什么意思？

答：实例化是一种奇怪的说法，指的是根据类创建对象。类型为类的对象是该类的实例。

## 10.15 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 10.15.1 测验

1. 什么是句点运算符，它的作用是什么？
2. 声明和定义哪个会分配内存？

3. 类声明是类的接口还是实现?
4. 公有数据成员和私有数据成员之间有何区别?
5. 成员函数可以是私有的吗?
6. 成员数据可以是公有的吗?
7. 如果创建两个 Cat 对象, 它们的 itsAge 成员数据可以有不同的值吗?
8. 类声明是以分号结尾吗? 类方法定义呢?
9. 在 Cat 类中, 不接受任何参数且返回类型为 void 的 Meow() 的函数头是什么样的?
10. 调用什么函数去初始化对象?

### 10.15.2 练习

1. 声明一个名为 Employee 的类, 它包含如下数据成员: age、yearsOfService 和 Salary。
2. 重写 Employee 类的声明, 使其数据成员为私有, 并提供获得和设置每个数据成员的公有存取器方法。
3. 编写一个程序, 使用 Employee 类来创建两个 Employees 对象, 设置它们的 age、YearOfService 和 Salary, 并打印它们的值。您还需要添加存取器方法的代码。
4. 继续练习 3, 给 Employee 类编写一个方法, 它报告雇员挣多少千美元 (四舍五入到最接近的千数)。

5. 修改 Employee 类, 以便能够在创建 Employee 对象时初始化 age、YearOfService 和 Salary。
6. 查错: 下面的声明有什么错误?

```
class Square
{
    public:
        int Side;
}
```

7. 查错: 为什么下面的类声明不是特别有用?

```
class Cat
{
    int GetAge() const;
private:
    int itsAge;
};
```

8. 查错: 编译器将在下面的代码中找到哪 3 处错误?

```
class TV
{
    public:
        void SetStation(int Station);
        int GetStation() const;
    private:
        int itsStation;
};

main()
{
    TV myTV;
    myTV.itsStation = 9;
    TV.SetStation(10);
    TV myOtherTv(2);
}
```





# 第 11 章

## 实现继承

继承是面向对象编程语言的最重要方面之一，正确使用继承可编写出设计良好、易于维护和可扩展的应用程序。

在本章中，您将学习：

- 什么是继承
- 如何使用继承从一个类派生出另一个类
- 什么是保护访问权限（protected access）及如何使用它
- 什么是虚方法
- 什么是私有继承

### 11.1 什么是继承

什么是狗？观察宠物时，您看到了什么？我看到 4 条腿和 1 张嘴。生物学家看到的是相互作用的器官网，物理学家看到的是原子和力，而分类学家看到的是犬类动物的代表。

现在令人感兴趣的是最后一种看法，狗是犬类的一种，而犬类动物是哺乳动物的一种，依次类推。分类学家将动物分成门、纲、目、科、属、种。

这种泛化/具体化层次结构建立了一种 is-a 关系。人是一种灵长目动物。这种关系无处不在：客货两用车是一种汽车，而汽车是一种交通工具。圣代冰激凌是一种甜食，而后者是一种食品。

当我们说一种东西是另外一种东西时，意味着它是后者的特例。也就是说，汽车是一种特殊的交通工具。

#### 11.1.1 继承和派生

狗继承（即自动获得）了哺乳动物的所有特点。狗是哺乳动物，因此我们知道它能够运动和呼吸。根据定义，所有哺乳动物都能够运动和呼吸。狗在哺乳动物的基础上增加了吠和摇尾巴等功能。

可以将狗分为役用犬、运动犬和猎犬。将运动犬分为捡拾东西的狗、哈巴狗等。每种这样的狗还可进一步细分，例如，捡拾东西的狗可细分为拉布拉多狗和加利福尼亚狗。

加利福尼亚猎狗是捡拾东西的狗，后者是一种运动犬，运动犬是一种狗，狗是一种哺乳动物，哺乳动物是一种动物，动物是一种生物。图 11.1 说明了这种层次结构。

C++ 允许您定义从另一个类派生出类来表示这种关系。派生是一种表示 is-a 关系的方式。您从类 Mammal 派生出新类 Dog。由于 Dog 类从 Mammal 类继承了运动功能，因此您不必显式说明狗能够运动。

在已有类的基础上添加了新功能的类被称为从原来的类派生而来。原来的类被称为新类的基类。

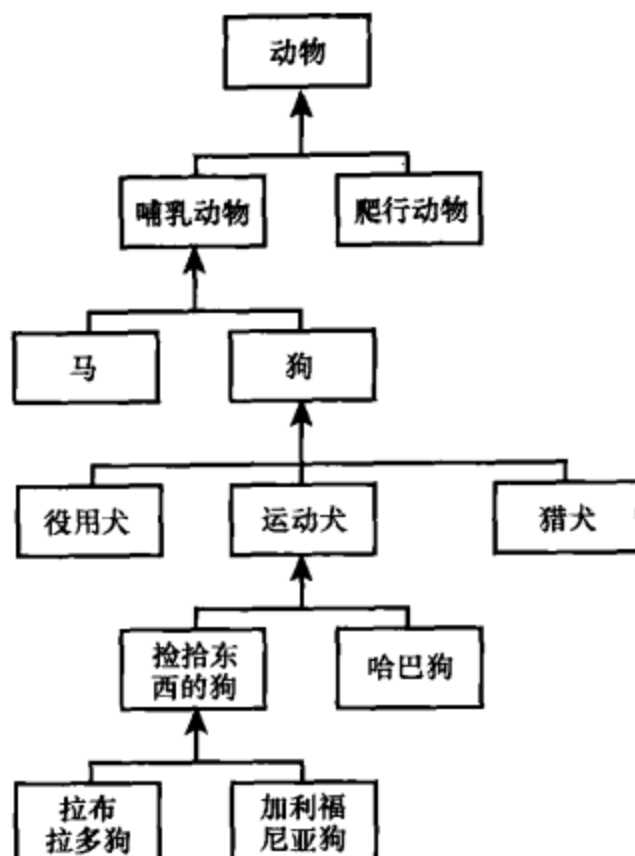


图 11.1 动物层次结构

如果 Dog 类从 Mammal 类派生而来，则 Mammal 类是 Dog 的基类。派生类是其基类的超集。狗在哺乳动物的基础上增加一些新特征，而 Dog 类在 Mammal 类的基础上增加了一些方法或数据。

通常，基类有多个派生类。由于狗、猫和马都是哺乳动物，因此它们对应的类都是从 Mammal 类派生而来的。

### 11.1.2 动物世界

为便于讨论派生和继承，本章将重点放在一些表示动物的类之间的关系上。假设有人请您设计一款孩子玩的游戏：模拟农场。

此时您将开发一组农场动物，包括马、奶牛、狗、猫和绵羊等。您将为这些类创建一些方法，以便它们能按孩子期望的方式行动，但就现在而言，将每个方法简化为只包含一条打印语句。

简化函数意味着只需编写能显示函数被调用的代码即可，将细节留待以后有时间去完成。读者可扩展本章提供的哺乳动物范例代码，让这些动物更为逼真。

读者将发现，使用动物的范例很容易理解，也很容易将这些概念用于其他领域。例如，编写 ATM 程序时，可能需要涉及支票账户，它是一种银行账户，而银行账户是一种账户。这类似于狗是哺乳动物，而哺乳动物是一种动物。

### 11.1.3 派生的语法

声明类时，可在类名后加上冒号 (:)、派生类型（公有或其他）和基类名来指出它是从哪个类派生而来的，格式如下：

```
class derivedClass : accessType baseClass
```

例如，要创建一个名为 Dog 的从 Mammal 类派生而来的新类，可以这样做：

```
class Dog : public Mammal
```

派生类型（accessType）将在本章后面讨论。就现在而言，总是使用 public。从中派生的类必须已经声明，否则将出现编译错误。程序清单 11.1 演示了如何声明从 Mammal 类派生而来的 Dog 类。

程序清单 11.1 简单继承

```
1: //Listing 11.1 Simple inheritance
2: #include <iostream>
3: using namespace std;
4:
5: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mammal
8: {
9:     public:
10:        // constructors
11:        Mammal();
12:        ~Mammal();
13:
14:        //accessors
15:        int GetAge() const;
16:        void SetAge(int);
17:        int GetWeight() const;
18:        void SetWeight();
19:
20:        //Other methods
21:        void Speak() const;
22:        void Sleep() const;
23:
24:
25:     protected:
26:         int itsAge;
27:         int itsWeight;
28: };
29:
30: class Dog : public Mammal
31: {
32:     public:
33:
34:        // Constructors
35:        Dog();
36:        ~Dog();
37:
38:        // Accessors
39:        BREED GetBreed() const;
40:        void SetBreed(BREED);
41:
42:        // Other methods
43:        WagTail();
44:        BegForFood();
45:
46:     protected:
47:         BREED itsBreed;
48: };
```

该程序没有输出，因为其中只有一组类声明，没有类的实现。虽然如此，还是有很多需要介绍的内容。

#### ▼ 分析：

第 7~28 行声明了 Mammal 类。注意，在这个例子中，Mammal 类没有继承任何类。但在现实生活中，哺乳动物确实是派生而来的，即哺乳动物是动物的一种。在 C++ 程序中，只能表示有关物体的部分信息，实际情况非常复杂，不可能表示全部信息，因此每个 C++ 层次结构都知识可用数据的有限表示。提供良好设计的技巧是，以相当忠实于实际情况的方式表示您关心的部分，而不增加不必要的复杂性。

层次结构必须从某个地方开始，这个程序从 Mammal 开始。由于这种决定，这个类中包含一些原本应该放在更高级基类中的成员变量。例如，所有动物都有年龄和体重，如果 Mammal 是从 Animal 派生而来的，将可以继承这些属性；然而，这些属性却出现在 Mammal 类中。

以后，如果添加了另一种也有这些属性的动物（如 Insect），可以将这些属性放到 Animal 类中，并将 Animal 作为 Mammal 和 Insect 的基类。这就是类层次结构是如何随时间推移而演变的。

为确保程序简单和易于管理, Mammal 类只包含 6 个方法: 4 个存取器方法以及 Speak() 和 Sleep()。如第 30 行所示, Dog 类是自 Mammal 类派生而来的, 这是因为类名 (Dog) 后面有冒号和基类名 (Mammal)。

每个 Dog 对象将有 3 个成员变量: itsAge、itsWeight 和 itsBreed。注意, Dog 的类声明中并没有成员变量 itsAge 和 itsWeight, Dog 对象从 Mammal 类继承了这些变量以及除复制构造函数、构造函数和析构函数外的所有方法。

## 11.2 私有和保护

读者可能注意到, 程序清单 11.1 的第 25 行和第 46 行使用了一个新的关键字 protected。以前, 类数据都被声明为私有的。然而, 私有成员在当前类之外是不能直接访问的, 在派生类中也如此。可以将变量 itsAge 和 itsWeight 声明为公有的, 但这并不理想, 因为您不希望其他类能够直接访问这些数据成员。

### 注意

有一种观点认为, 应将所有成员数据声明为私有, 而不是保护的; 这是 C++ 创始人 Bjarne Stroustrup 在其 *The Design and Evolution of C++* (1994) 一书中指出的。受保护的方法通常不被认为是有问题, 而且可能很有用。

您想要的派生类型是: 使这些数据对当前类及其派生类来说是可见的。这种派生类型为保护 (protected)。保护型数据成员和函数对派生类来说完全可见, 但对其他类来说是私有的。

总共有 3 种访问限定符: 公有 (public)、保护 (protected) 和私有 (private)。如果在函数中声明了一个对象, 则它可以访问所有的公有成员数据和函数。而成员函数可以访问其所属类的所有私有数据成员和函数, 也可以访问任何从其所属类派生而来的类的保护数据成员和函数。

因此, 函数 Dog::WagTail() 可以访问私有数据 itsBreed 以及 Mammal 类的保护数据 itsAge 和 itsWeight。

即使继承层次结构中, Mammal 类和 Dog 类之间有其他类 (如 DomesticAnimals), Dog 类仍能够访问 Mammal 类的保护成员, 条件是这些其他类都采用公有继承。私有继承将在本章后面讨论。

程序清单 11.2 演示了如何创建 Dog 对象以及如何访问其数据和函数。

程序清单 11.2 使用派生对象

```

1: //Listing 11.2 Using a derived object
2: #include <iostream>
3: using std::cout;
4: using std::endl;
5:
6: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
7:
8: class Mammal
9: {
10: public:
11:     // constructors
12:     Mammal():itsAge(2), itsWeight(5){}
13:     ~Mammal(){}
14:
15:     //accessors
16:     int GetAge() const { return itsAge; }
17:     void SetAge(int age) { itsAge = age; }
18:     int GetWeight() const { return itsWeight; }
19:     void SetWeight(int weight) { itsWeight = weight; }
20:
21:     //Other methods
22:     void Speak()const { cout << "Mammal sound!\n"; }
23:     void Sleep()const { cout << "shhh. I'm sleeping.\n"; }
24:

```

```
25:     protected:
26:         int itsAge;
27:         int itsWeight;
28:     };
29:
30:     class Dog : public Mammal
31:     {
32:     public:
33:
34:         // Constructors
35:         Dog():itsBreed(GOLDEN){}
36:         ~Dog(){}
37:
38:         // Accessors
39:         BREED GetBreed() const { return itsBreed; }
40:         void SetBreed(BREED breed) { itsBreed = breed; }
41:
42:         // Other methods
43:         void WagTail() const { cout << "Tail wagging...\n"; }
44:         void BegForFood() const { cout << "Begging for food...\n"; }
45:
46:     private:
47:         BREED itsBreed;
48:     };
49:
50:     int main()
51:     {
52:         Dog Fido;
53:         Fido.Speak();
54:         Fido.WagTail();
55:         cout << "Fido is " << Fido.GetAge() << " years old" << endl;
56:         return 0;
57:     }
```

#### ▼ 输出:

```
Mammal sound!
Tail wagging...
Fido is 2 years old
```

#### ▼ 分析:

第 8~28 行声明了 Mammal 类 (为节省篇幅, 其所有函数都是内联的)。第 30~48 行将 Dog 类声明为 Mammal 的派生类。通过这些声明, 所有 Dog 对象都有年龄、体重和品种。正如前面指出的, 年龄和体重是从基类 Mammal 那里继承而来的。

第 52 行声明了一个 Dog 对象: Fido。Fido 继承了 Mammal 的所有属性, 同时具有 Dog 的所有属性。因此, Fido 不但知道如何 WagTail(), 还知道如何 Speak() 和 Sleep()。第 55 行成功地调用了基类中的存取器方法 GetAge()。

## 11.3 构造函数和析构函数的继承性

Dog 对象是 Mammal 对象, 这是 is-a 关系的本质。

创建 Fido 时, 首先调用基类的构造函数, 创建一个 Mammal 对象; 然后调用 Dog 的构造函数, 完成 Dog 对象的创建。由于创建 Fido 时没有提供参数, 因此将调用 Mammal 和 Dog 的默认构造函数。在 Fido 创建好之前, 它是不存在的, 这就意味着必须创建其 Mammal 部分和 Dog 部分。因此, 必须调用这两个类的构造函数。

Fido 被销毁时, 首先调用 Dog 的析构函数, 然后为 Fido 的 Mammal 部分调用析构函数。每个析构函数都提供了在其对应部分被销毁后执行清理工作的机会。别忘了, 在 Dog 对象被销毁后进行清理! 程序清单 11.3 演示了对构造函数和析构函数的调用。



## 程序清单 11.3 调用构造函数和析构函数

```
1: //Listing 11.3 Constructors and destructors called.
2: #include <iostream>
3: using namespace std;
4: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
5:
6: class Mammal
7: {
8:     public:
9:         // constructors
10:        Mammal();
11:        ~Mammal();
12:
13:        //accessors
14:        int GetAge() const { return itsAge; }
15:        void SetAge(int age) { itsAge = age; }
16:        int GetWeight() const { return itsWeight; }
17:        void SetWeight(int weight) { itsWeight = weight; }
18:
19:        //Other methods
20:        void Speak() const { cout << "Mammal sound!\n"; }
21:        void Sleep() const { cout << "shhh. I'm sleeping.\n"; }
22:
23:    protected:
24:        int itsAge;
25:        int itsWeight;
26: };
27:
28: class Dog : public Mammal
29: {
30:     public:
31:
32:        // Constructors
33:        Dog();
34:        ~Dog();
35:
36:        // Accessors
37:        BREED GetBreed() const { return itsBreed; }
38:        void SetBreed(BREED breed) { itsBreed = breed; }
39:
40:        // Other methods
41:        void WagTail() const { cout << "Tail wagging...\n"; }
42:        void BegForFood() const { cout << "Begging for food...\n"; }
43:
44:    private:
45:        BREED itsBreed;
46: };
47:
48: Mammal::Mammal():
49:     itsAge(3),
50:     itsWeight(5)
51: {
52:     std::cout << "Mammal constructor... " << endl;
53: }
54:
55: Mammal::~Mammal()
56: {
57:     std::cout << "Mammal destructor... " << endl;
58: }
59:
60: Dog::Dog():
61:     itsBreed(GOLDEN)
62: {
63:     std::cout << "Dog constructor... " << endl;
64: }
65:
66: Dog::~Dog()
67: {
68:     std::cout << "Dog destructor... " << endl;
69: }
70: int main()
```

```

71: {
72:     Dog Fido;
73:     Fido.Speak();
74:     Fido.WagTail();
75:     std::cout << "Fido is " << Fido.GetAge() << " years old" << endl;
76:     return 0;
77: }

```

#### ▼ 输出:

```

Mammal constructor...
Dog constructor...
Mammal sound!
Tail wagging...
Fido is 3 years old
Dog destructor...
Mammal destructor...

```

#### ▼ 分析:

除第 48~69 行的构造函数和析构函数在被调用时将信息打印到屏幕上外,程序清单 11.3 和程序清单 11.2 相同。首先调用 Mammal 的构造函数,然后调用 Dog 的构造函数。至此, Dog 对象被创建好,才可以调用其方法。

当 Fido 不再在作用域中时, Dog 的析构函数首先被调用,然后 Mammal 的析构函数被调用。

### 向基类的构造函数传递参数

您可能想在基类构造函数中初始化某些值。例如,您可能想重载 Mammal 的构造函数使其接受年龄作为参数,并重载 Dog 的构造函数使其接受品种作为参数。如何确保年龄和体重参数被传递给正确的 Mammal 构造函数呢? 如果 Dog 想初始化体重而 Mammal 不想该如何呢?

通过在派生类的构造函数名后加上冒号、基类名和基类构造函数需要的参数,可以在初始化期间对基类进行初始化。程序清单 11.4 说明了这一点。

#### 程序清单 11.4 在派生类中重载构造函数

```

1: //Listing 11.4 Overloading constructors in derived classes
2: #include <iostream>
3: using namespace std;
4:
5: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mammal
8: {
9:     public:
10:        // constructors
11:        Mammal();
12:        Mammal(int age);
13:        ~Mammal();
14:
15:        //accessors
16:        int GetAge() const { return itsAge; }
17:        void SetAge(int age) { itsAge = age; }
18:        int GetWeight() const { return itsWeight; }
19:        void SetWeight(int weight) { itsWeight = weight; }
20:
21:        //Other methods
22:        void Speak() const { cout << "Mammal sound!\n"; }
23:        void Sleep() const { cout << "shhh. I'm sleeping.\n"; }
24:
25:
26:     protected:
27:        int itsAge;
28:        int itsWeight;
29: };
30:

```

```
31: class Dog : public Mammal
32: {
33:     public:
34:
35:         // Constructors
36:         Dog();
37:         Dog(int age);
38:         Dog(int age, int weight);
39:         Dog(int age, BREED breed);
40:         Dog(int age, int weight, BREED breed);
41:         ~Dog();
42:
43:         // Accessors
44:         BREED GetBreed() const { return itsBreed; }
45:         void SetBreed(BREED breed) { itsBreed = breed; }
46:
47:         // Other methods
48:         void WagTail() const { cout << "Tail wagging...\n"; }
49:         void BegForFood() const { cout << "Begging for food...\n"; }
50:
51:     private:
52:         BREED itsBreed;
53: };
54:
55: Mammal::Mammal():
56:     itsAge(1),
57:     itsWeight(5)
58: {
59:     cout << "Mammal constructor..." << endl;
60: }
61:
62: Mammal::Mammal(int age):
63:     itsAge(age),
64:     itsWeight(5)
65: {
66:     cout << "Mammal(int) constructor..." << endl;
67: }
68:
69: Mammal::~~Mammal()
70: {
71:     cout << "Mammal destructor..." << endl;
72: }
73:
74: Dog::Dog():
75:     Mammal(),
76:     itsBreed(GOLDEN)
77: {
78:     cout << "Dog constructor..." << endl;
79: }
80:
81: Dog::Dog(int age):
82:     Mammal(age),
83:     itsBreed(GOLDEN)
84: {
85:     cout << "Dog(int) constructor..." << endl;
86: }
87:
88: Dog::Dog(int age, int weight):
89:     Mammal(age),
90:     itsBreed(GOLDEN)
91: {
92:     itsWeight = weight;
93:     cout << "Dog(int, int) constructor..." << endl;
94: }
95:
96: Dog::Dog(int age, int weight, BREED breed):
97:     Mammal(age),
98:     itsBreed(breed)
99: {
100:     itsWeight = weight;
101:     cout << "Dog(int, int, BREED) constructor..." << endl;
102: }
```

```

103:
104: Dog::Dog(int age, BREED breed):
105:     Mammal(age),
106:     itsBreed(breed)
107: {
108:     cout << "Dog(int, BREED) constructor..." << endl;
109: }
110:
111: Dog::~Dog()
112: {
113:     cout << "Dog destructor..." << endl;
114: }
115: int main()
116: {
117:     Dog Fido;
118:     Dog rover(5);
119:     Dog buster(6,8);
120:     Dog yorkie (3,GOLDEN);
121:     Dog dobbie (4,20,DOBERMAN);
122:     Fido.Speak();
123:     rover.WagTail();
124:     cout << "Yorkie is " << yorkie.GetAge()
125:         << " years old" << endl;
126:     cout << "Dobbie weighs ";
127:     cout << dobbie.GetWeight() << " pounds" << endl;
128:     return 0;
129: }

```

**注意**

为便于分析时引用，给输出加上了行号。

**▼ 输出:**

```

1: Mammal constructor...
2: Dog constructor...
3: Mammal(int) constructor...
4: Dog(int) constructor...
5: Mammal(int) constructor...
6: Dog(int, int) constructor...
7: Mammal(int) constructor...
8: Dog(int, BREED) constructor...
9: Mammal(int) constructor...
10: Dog(int, int, BREED) constructor...
11: Mammal sound!
12: Tail wagging...
13: Yorkie is 3 years old.
14: Dobbie weighs 20 pounds.
15: Dog destructor...
16: Mammal destructor...
17: Dog destructor...
18: Mammal destructor...
19: Dog destructor...
20: Mammal destructor...
21: Dog destructor...
22: Mammal destructor...
23: Dog destructor...
24: Mammal destructor...

```

**▼ 分析:**

在程序清单 11.4 中，第 12 行重载了 Mammal 的构造函数，使之接受一个 int 参数（Mammal 的年龄）。第 62~67 行为该构造函数的实现，它用传递给构造函数的值来初始化 itsAge 进行初始化，并将 itsWeight 初始化为 5。

第 36~40 行重载了 5 个 Dog 构造函数。第 1 个是默认构造函数。在第 37 行，第 2 个接受年龄作为参数，与 Mammal 的构造函数接受的参数相同。第 3 个构造函数接受年龄和体重作为参数。第 4 个构造函数接受年龄和品种作为参数。而第 5 个接受年龄、体重和品种作为参数。

从第 74 行开始, 是 Dog 的默认构造函数的代码, 其中有些新内容。从第 75 行可知, 该构造函数被调用时, 它将调用 Mammal 的默认构造函数。虽然并不是非得这样做, 但它表明要调用不接受任何参数的基类构造函数。在任何情况下, 都将调用基类构造函数, 但通过显式地调用, 可使您的意图更明确。

接受一个 int 参数的 Dog 构造函数的实现位于第 81~86 行。在其初始化阶段 (第 82~83 行), Dog 使用传入的参数初始化基类, 然后初始化品种。

另一个 Dog 构造函数位于第 88~94 行, 它接受两个参数。同样, 它在 89 行通过调用合适的构造函数来初始化基类, 但这次也将体重参数赋给基类的 itsWeight 变量。注意, 不能在初始化阶段给基类的 itsWeight 变量赋值, 因为 Mammal 类没有接受这种参数的构造函数, 因此必须在 Dog 的构造函数体中完成这项工作。

请读者分析其他的构造函数, 确保自己理解了它们的工作原理。请注意哪些成员变量被初始化, 哪些成员变量必须到构造函数体中进行赋值。

为方便在分析中引用, 给输出加上了行号。前两行输出表明, Fido 是使用默认构造函数实例化的。

输出的第 3 行和第 4 行表示 rover 被创建, 第 5 行和第 6 行表示 buster 被创建。注意, 调用的是接受一个 int 参数的 Mammal 构造函数, 但调用的 Dog 构造函数是接受两个 int 参数的构造函数。

所有对象被创建后, 使用了这些对象, 然后它们不再在作用域中。每个对象被销毁时, 首先调用 Dog 的析构函数, 然后调用 Mammal 的析构函数, 每个析构函数都被调用了 5 次。

## 11.4 覆盖基类函数

Dog 对象可以访问 Mammal 类的所有成员函数, 也可以访问 Dog 类新增的任何数据成员和函数, 如 WagTail()。派生类还可以覆盖基类函数。覆盖基类函数意味着在派生类中修改其实现。

在派生类中创建一个返回值和特征标与基类成员函数相同但实现不同的函数时, 被称为覆盖该函数。在这种情况下, 创建一个派生类对象后, 便可以调用正确的函数。

覆盖函数时, 特征标必须与基类中被覆盖的函数相同。特征标指的是函数原型中除返回类型外的内容, 即函数名、参数列表和可能用到的关键字 const。

程序清单 11.5 说明了在 Dog 类中覆盖 Mammal 的 Speak() 方法后将发生的情况。为节省篇幅, 删除了这些类的存取器函数。

程序清单 11.5 在派生类中覆盖基类方法

```
1: //Listing 11.5 Overriding a base class method in a derived class
2: #include <iostream>
3: using std::cout;
4:
5: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mammal
8: {
9:     public:
10:        // constructors
11:        Mammal() { cout << "Mammal constructor...\n"; }
12:        ~Mammal() { cout << "Mammal destructor...\n"; }
13:
14:        //Other methods
15:        void Speak()const { cout << "Mammal sound!\n"; }
16:        void Sleep()const { cout << "shhh. I'm sleeping.\n"; }
17:
18:    protected:
19:        int itsAge;
20:        int itsWeight;
21: };
22:
23: class Dog : public Mammal
24: {
```



```
25: public:
26:     // Constructors
27:     Dog(){ cout << "Dog constructor...\n"; }
28:     ~Dog(){ cout << "Dog destructor...\n"; }
29:
30:     // Other methods
31:     void WagTail() const { cout << "Tail wagging...\n"; }
32:     void BegForFood() const { cout << "Begging for food...\n"; }
33:     void Speak() const { cout << "Woof!\n"; }
34:
35: private:
36:     BREED itsBreed;
37: };
38:
39: int main()
40: {
41:     Mammal bigAnimal;
42:     Dog Fido;
43:     bigAnimal.Speak();
44:     Fido.Speak();
45:     return 0;
46: }
```

#### ▼ 输出:

```
Mammal constructor...
Mammal constructor...
Dog constructor...
Mammal sound!
Woof!
Dog destructor...
Mammal destructor...
Mammal destructor...
```

#### ▼ 分析:

在 Mammal 类中, 第 15 行定义了一个名为 Speak() 的方法。第 23~37 行声明了 Dog 类, 它是从 Mammal 派生而来的 (第 23 行), 因此能够访问该 Speak() 方法。然而, 在 33 行, Dog 类覆盖了 Speak() 方法, 导致 Speak() 方法被调用时, Dog 对象打印 “Woof!”。

在 main() 中, 第 41 行创建了 Mammal 对象 bigAnimal, 这导致 Mammal 的构造函数被调用, 打印第一行输出。第 42 行创建了 Dog 对象 fido, 这导致首先调用 Mammal 的构造函数, 然后调用 Dog 的构造函数, 它们打印接下来的两行输出。

第 43 行通过 Mammal 对象调用 Speak() 方法, 然后第 44 行通过 Dog 对象调用 Speak() 方法。输出表明, 调用了正确的方法: bigAnimal 发出哺乳动物叫声 (打印 “Mammal Sound!”), Fido 发出低叫声 (打印 “Woof!”)。最后, 这两个对象不再在作用域中, 因此析构函数被调用。

#### 重载和覆盖

这两个术语类似, 功能也相似。重载方法时, 创建多个名称相同但特征标不同的方法; 覆盖方法时, 在派生类中创建一个名称和特征标都与基类方法相同的方法。

### 11.4.1 隐藏基类的方法

在前一个程序清单中, Dog 类的 Speak() 方法隐藏了基类的 Speak() 方法。这正是我们希望的, 但可能有意想不到的结果。如果 Mammal 有一个被重载的方法 Move(), 而 Dog 覆盖了该方法, 则 Dog 的 Move() 方法将隐藏 Mammal 中所有名称为 Move 的方法。

如果 Mammal 重载了 3 个 Move() 方法: 第一个不接受任何参数, 第二个接受一个 int 参数, 第三个接受一个 int 参数和一个方向参数, 且 Dog 只覆盖了不接受任何参数的 Move() 方法, 将难以通过

Dog 对象访问其他两个 Move() 方法。程序清单 11.6 说明了这种问题。

程序清单 11.6 隐藏方法

```
1: //Listing 11.6 Hiding methods
2: #include <iostream>
3: using std::cout;
4:
5: class Mammal
6: {
7:     public:
8:         void Move() const { cout << "Mammal move one step.\n"; }
9:         void Move(int distance) const
10:        {
11:            cout << "Mammal move ";
12:            cout << distance << " steps.\n";
13:        }
14:     protected:
15:         int itsAge;
16:         int itsWeight;
17: };
18:
19: class Dog : public Mammal
20: {
21:     public:
22:         // You might receive a warning that you are hiding a function!
23:         void Move() const { cout << "Dog move 5 steps.\n"; }
24: };
25:
26: int main()
27: {
28:     Mammal bigAnimal;
29:     Dog Fido;
30:     bigAnimal.Move();
31:     bigAnimal.Move(2);
32:     Fido.Move();
33:     // Fido.Move(10);
34:     return 0;
35: }
```

#### ▼ 输出:

```
Mammal move one step.
Mammal move 2 steps.
Dog move 5 steps.
```

#### ▼ 分析:

所有多余的方法和数据都从这些类中删除了。在第 8 行和第 9 行, Mammal 类声明了重载的 Move() 方法。在第 23 行, Dog 覆盖了接受任何参数的 Move() 方法。第 30~32 行调用了这些方法, 程序运行时输出反映了这一点。

然而, 第 33 行将导致编译错误, 因此被注释掉了。覆盖基类的某个方法后, 就不能通过派生类对象使用任何同名的基类方法。如果 Dog 类没有覆盖不接受任何参数的 Move() 方法, 它将可以调用 Move(int) 方法, 但既然覆盖了, 如果要使用这两个版本的 move() 方法, 必须将它们都覆盖。否则将隐藏没有覆盖的方法。与该规则类似的情形是, 如果您提供了任何构造函数, 编译器将不会提供默认构造函数。

规则是这样的: 覆盖任一个重载方法后, 该方法的其他所有版本都将被隐藏; 如果不希望它们被隐藏, 必须对其进行覆盖。

一种常见的错误是, 在本想覆盖一个基类方法时, 由于忘记包含关键字 const, 而将其隐藏了。const 是特征标的一部分, 省略它将改变特征标, 导致方法被隐藏而不是被覆盖。

#### 注意

下一节将讨论虚方法。通过覆盖虚方法可支持多态, 但隐藏虚方法将破坏多态。稍后将更详细地介绍这一点。

## 11.4.2 调用基类方法

覆盖基类方法后，仍可以通过限定方法名来调用它——在方法名前加上基类名和两个冒号：

```
baseClass::Method()
```

要调用 Mammal 类的 Move() 方法，可以这样做：

```
Mammal::Move().
```

可以像使用其他方法名一样使用这些被限定的方法名。可以像下面这样修改清单 11.6 的第 33 行，使其能够通过编译：

```
Fido.Mammal::Move(10);
```

这显式地调用 Mammal 的方法。程序清单 11.7 演示了这种概念。

程序清单 11.7 调用被覆盖的基类方法

```
1: //Listing 11.7 Calling a base method from a overridden method.
2: #include <iostream>
3: using namespace std;
4:
5: class Mammal
6: {
7:     public:
8:         void Move() const { cout << "Mammal move one step\n"; }
9:         void Move(int distance) const
10:        {
11:            cout << "Mammal move " << distance;
12:            cout << " steps." << endl;
13:        }
14:
15:     protected:
16:         int itsAge;
17:         int itsWeight;
18: };
19:
20: class Dog : public Mammal
21: {
22:     public:
23:         void Move() const;
24: };
25:
26: void Dog::Move() const
27: {
28:     cout << "In dog move...\n";
29:     Mammal::Move(3);
30: }
31:
32: int main()
33: {
34:     Mammal bigAnimal;
35:     Dog Fido;
36:     bigAnimal.Move(2);
37:     Fido.Mammal::Move(6);
38:     return 0;
39: }
```

### ▼ 输出：

```
Mammal move 2 steps.
Mammal move 6 steps.
```

### ▼ 分析：

第 35 行创建了 Mammal 对象 bigAnimal，第 36 行创建了 Dog 对象 Fido。第 37 行 Mammal 类中接受一个参数的 Move() 方法。

程序员想对 Dog 对象调用方法 Move(int)，但存在一个问题。Dog 覆盖了不接受任何参数的 Move()

方法，但没有对其进行重载使其接受一个 `int` 参数，即没有提供接受一个 `int` 参数的版本。第 37 行通过显式地调用基类的 `Move(int)` 方法解决了这种问题。

#### 提示

通过使用 `::` 来调用祖先的类方法时，如果在继承层次结构中，在祖先和后代之间插入了新类，后代将跳过这些中间类，从而遗漏对中间类实现的重要功能的调用。

#### 应该

应通过派生来扩展经过测试的类的功能。  
通过覆盖基类方法来改变派生类中某些函数的行为。

#### 不应该

不要通过修改函数特征标来隐藏基类函数。  
别忘了 `const` 是特征标的组成部分。  
别忘了返回类型不是特征标的组成部分。

## 11.5 虚方法

本章强调了这样一个事实：`Dog` 对象是一个 `Mammal` 对象。到目前为止，这仅意味着 `Dog` 对象继承了其基类的属性（数据）和功能（方法）。然而，在 C++ 中，`is-a` 关系要比这深入得多。

C++ 扩展了其多态性，允许将派生类对象赋给指向基类的指针。因此，可以这样编写代码：

```
Mammal* pMammal = new Dog;
```

上述代码在堆中创建了一个新的 `Dog` 对象，并返回一个指向该对象的指针，然后将该指针赋给一个 `Mammal` 指针。之所以可以这样做，是因为狗是一种哺乳动物。

#### 注意

这是多态的本质。例如，可创建很多类型的窗口，包括对话框、可滚动窗口和列表框，然后给每种窗口定义一个虚方法 `draw()`。通过创建一个窗口指针，并将对话框和其他派生类对象赋给指针，就可以调用方法 `draw()`，而不用考虑运行时指针指向的实际对象类型。程序将调用正确的 `draw()` 方法。

然后可以使用这个指针来调用 `Mammal` 类的任何方法。您希望在调用被 `Dog` 覆盖的方法时，将调用正确的函数。虚函数让您能够做到这一点。要创建虚函数，可在函数声明前加上关键字 `virtual`。程序清单 11.8 演示了虚函数的工作原理以及使用非虚方法时将发生的情况。

程序清单 11.8 使用虚方法

```
1: //Listing 11.8 Using virtual methods
2: #include <iostream>
3: using std::cout;
4:
5: class Mammal
6: {
7:     public:
8:         Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
9:         virtual ~Mammal() { cout << "Mammal destructor...\n"; }
10:        void Move() const { cout << "Mammal move one step\n"; }
11:        virtual void Speak() const { cout << "Mammal speak!\n"; }
12:
13:     protected:
14:         int itsAge;
15: };
16:
17: class Dog : public Mammal
18: {
19:     public:
20:         Dog() { cout << "Dog Constructor...\n"; }
21:         virtual ~Dog() { cout << "Dog destructor...\n"; }
22:         void WagTail() { cout << "Wagging Tail...\n"; }
```

```
23:     void Speak()const { cout << "Woof!\n"; }
24:     void Move()const { cout << "Dog moves 5 steps...\n"; }
25: };
26:
27: int main()
28: {
29:     Mammal *pDog = new Dog;
30:     pDog->Move();
31:     pDog->Speak();
32:
33:     return 0;
34: }
```

### ▼ 输出:

```
Mammal constructor...
Dog Constructor...
Mammal move one step
Woof!
```

### ▼ 分析:

第 11 行给 Mammal 类提供了一个虚方法: `speak()`。这个类的设计者指出,他希望这个类最终会是另一个类的基类。派生类可能想覆盖这个函数。

第 29 行创建了一个 Mammal 指针 (`pDog`),但将一个新 Dog 对象的地址赋给它。由于狗是哺乳动物,因此这种赋值是合法的。然后第 30 行使用这个指针来调用函数 `Move()`。由于编译器知道 `pDog` 是一个 Mammal 指针,因此在 Mammal 类中查找 `Move()` 方法。从第 10 行可知,这是一个标准的、非虚方法。

第 31 行通过该指针调用 `Speak()` 函数。由于 `Speak()` 是一个虚方法,因此调用 Dog 中的 `Speak()` 函数。

这几乎是在变魔术!对调用函数来说,它有一个 Mammal 指针,但这里却调用了 Dog 的方法。事实上,如果有一个 Mammal 指针数组,其中每个指针都指向不同的 Mammal 子类对象,则可以依次通过这些指针来调用虚方法,这将调用正确的函数。程序清单 11.9 说明了这一点。

### 程序清单 11.9 依次调用多个虚方法

```
1: //Listing 11.9 Multiple virtual functions called in turn
2: #include <iostream>
3: using namespace std;
4:
5: class Mammal
6: {
7:     public:
8:         Mammal():itsAge(1) { }
9:         virtual ~Mammal() { }
10:        virtual void Speak() const { cout << "Mammal speak!\n"; }
11:
12:     protected:
13:         int itsAge;
14: };
15:
16: class Dog : public Mammal
17: {
18:     public:
19:         void Speak()const { cout << "Woof!\n"; }
20: };
21:
22: class Cat : public Mammal
23: {
24:     public:
25:         void Speak()const { cout << "Meow!\n"; }
26: };
27:
28:
29: class Horse : public Mammal
```



```
30: {
31:     public:
32:         void Speak()const { cout << "Winnie!\n"; }
33: };
34:
35: class Pig : public Mammal
36: {
37:     public:
38:         void Speak()const { cout << "Oink!\n"; }
39: };
40:
41: int main()
42: {
43:     Mammal* theArray[5];
44:     Mammal* ptr;
45:     int choice, i;
46:     for ( i = 0; i<5; i++)
47:     {
48:         cout << "(1)dog (2)cat (3)horse (4)pig: ";
49:         cin >> choice;
50:         switch (choice)
51:         {
52:             case 1: ptr = new Dog;
53:                 break;
54:             case 2: ptr = new Cat;
55:                 break;
56:             case 3: ptr = new Horse;
57:                 break;
58:             case 4: ptr = new Pig;
59:                 break;
60:             default: ptr = new Mammal;
61:                 break;
62:         }
63:         theArray[i] = ptr;
64:     }
65:     for (i=0;i<5;i++)
66:         theArray[i]->Speak();
67:     return 0;
68: }
```

#### ▼ 输出:

```
(1)dog (2)cat (3)horse (4)pig: 1
(1)dog (2)cat (3)horse (4)pig: 2
(1)dog (2)cat (3)horse (4)pig: 3
(1)dog (2)cat (3)horse (4)pig: 4
(1)dog (2)cat (3)horse (4)pig: 5
Woof!
Meow!
Whinny!
Oink!
Mammal speak!
```

#### ▼ 分析:

这个简化的程序只提供了每个类最基本的功能,它以最简洁的形式演示了虚方法。声明了 4 个类: Dog、Cat、Horse 和 Pig,它们都是从 Mammal 类派生而来的。

第 10 行将 Mammal 的 Speak() 函数声明为虚方法。在第 19、25、32 和 38 行,4 个派生类覆盖了 Speak() 函数的实现。

在第 47~64 行,程序循环 5 次,每次都提示用户选择要创建哪种对象,然后第 50~62 行的 switch 语句将一个指向这种对象的指针加入到数组中。

在该程序编译时,无法知道将创建什么类型的对象,因此也无法知道将调用哪个 Speak() 方法。ptr 指向的对象是在运行阶段确定的,这被称为动态绑定或运行阶段绑定,与此相对的是静态绑定或编译阶段绑定。

FAQ

如果在基类中将一个成员方法标记为虚方法，还需要在派生类中将它标记为虚方法吗？

答：不需要。方法被声明为虚方法后，如果在派生类覆盖它，它仍是虚方法。在派生类中继续将方法标记为虚方法是个不错的主意（虽然不是必须这样做），这将使代码更容易理解。

### 11.5.1 虚函数的工作原理

创建派生对象（如 Dog 对象）时，首先调用基类的构造函数，然后调用派生类的构造函数。图 11.2 说明了 Dog 对象被创建后的情景。注意，Mammal 部分和 Dog 部分在内存中是相邻的。

在类中创建虚方法后，这个类的对象必须跟踪虚方法。很多编译器创建了虚函数表（v-table）。每个类都有一个虚函数表，每个类对象都有一个指向虚函数表的指针（vptr 或 v-pointer）。

虽然实现方式不同，但所有编译器都必须完成这项工作。每个对象的 vptr 都指向 v-table，而对于每个虚方法，v-table 都包含一个指向它的指针。创建 Dog 的 Mammal 部分时，vptr 被初始化为指向 v-table 的正确部分，如图 11.3 所示。

当 Dog 的构造函数被调用时，添加对象的 Dog 部分，并调整 vptr 指针使其指向 Dog 类中覆盖的虚方法（如果有的话），如图 11.4 所示。

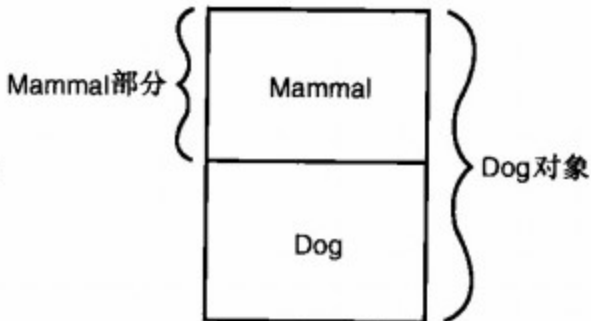


图 11.2 创建后的 Dog 对象

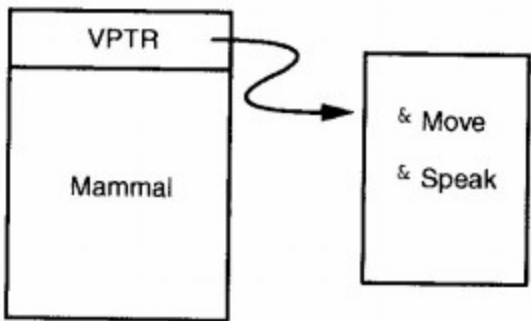


图 11.3 Mammal 的 v-table

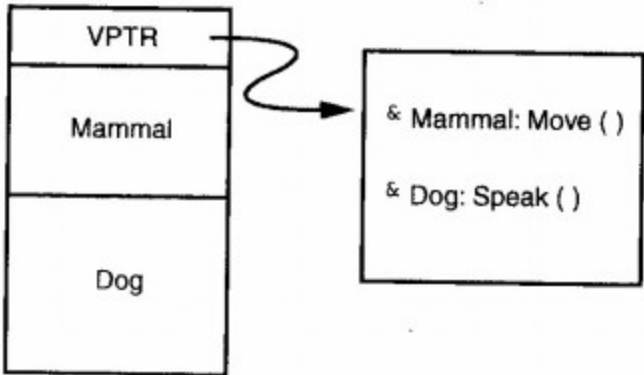


图 11.4 Dog 的 v-table

使用 Mammal 指针时，vptr 将根据 Mammal 指针指向的对象的实际类型指向正确的函数。这样，调用函数 Speak() 时，将调用正确的函数。

### 11.5.2 通过基类指针访问派生类的方法

前面介绍过，通过基类指针可以访问派生类的虚方法。如果派生类有一个基类没有的方法，可以像访问虚方法那样通过基类指针来访问它吗？由于只有派生类有该方法，因此不存在名称冲突。

如果 Dog 有一个 WagTail() 方法，但 Mammal 没有，则不能通过 Mammal 指针来访问该方法。由于 WagTail() 不是虚方法，且 Mammal 没有这种方法，因此如果没有 Dog 对象或 Dog 指针，将不能访问它。

虽然可以将 Mammal 指针强制转换为 Dog 指针，但如果 Mammal 不是一个 Dog，这样做将是不安全的。虽然通过强制类型转换，可以将 Mammal 指针转换为 Dog 指针，但有一种更好、更安全的调用

WagTail()函数的方法。C++不赞成强制类型转换，因为这样容易出错。

### 11.5.3 切除

仅当通过指针和引用进行调用时，才能发挥虚方法的魔力，按值传递对象时将不能发挥虚方法的魔力。程序清单 11.10 说明了这种问题。

程序清单 11.10 按值传递时的数据切除 (slicing)

```

1: //Listing 11.10 Data slicing with passing by value
2: #include <iostream>
3: using namespace std;
4:
5: class Mammal
6: {
7:     public:
8:         Mammal():itsAge(1) { }
9:         virtual ~Mammal() { }
10:        virtual void Speak() const { cout << "Mammal speak!\n"; }
11:
12:    protected:
13:        int itsAge;
14: };
15:
16: class Dog : public Mammal
17: {
18:     public:
19:         void Speak()const { cout << "Woof!\n"; }
20: };
21:
22: class Cat : public Mammal
23: {
24:     public:
25:         void Speak()const { cout << "Meow!\n"; }
26: };
27:
28: void ValueFunction (Mammal);
29: void PtrFunction (Mammal*);
30: void RefFunction (Mammal&);
31: int main()
32: {
33:     Mammal* ptr=0;
34:     int choice;
35:     while (1)
36:     {
37:         bool fQuit = false;
38:         cout << "(1)dog (2)cat (0)Quit: ";
39:         cin >> choice;
40:         switch (choice)
41:         {
42:             case 0: fQuit = true;
43:                     break;
44:             case 1: ptr = new Dog;
45:                     break;
46:             case 2: ptr = new Cat;
47:                     break;
48:             default: ptr = new Mammal;
49:                     break;
50:         }
51:         if (fQuit == true)
52:             break;
53:         PtrFunction(ptr);
54:         RefFunction(*ptr);
55:         ValueFunction(*ptr);
56:     }
57:     return 0;
58: }
59:
60: void ValueFunction (Mammal MammalValue)

```

```

61: {
62:     MammalValue.Speak();
63: }
64:
65: void PtrFunction (Mammal * pMammal)
66: {
67:     pMammal->Speak();
68: }
69:
70: void RefFunction (Mammal & rMammal)
71: {
72:     rMammal.Speak();
73: }

```

### ▼ 输出:

```

(1)dog (2)cat (0)Quit: 1
Woof
Woof
Mammal Speak!
(1)dog (2)cat (0)Quit: 2
Meow!
Meow!
Mammal Speak!
(1)dog (2)cat (0)Quit: 0

```

### ▼ 分析:

第 4~26 行声明了简化的 Mammal、Dog 和 Cat 类。声明了 3 个函数: PtrFunction()、RefFunction() 和 ValueFunction(), 它们分别接受 Mammal 指针、Mammal 引用和 Mammal 对象作为参数。从第 60~73 行可知, 这 3 个函数都做同一件事: 调用 Speak() 方法。

程序提示用户选择 Dog 或 Cat, 根据用户的选择, 第 44 行或第 46 行创建一个指向相应类型的对象指针。

输出的第一行表明, 用户选择了 Dog。第 44 行在自由存储区中创建一个 Dog 对象。然后通过指针 (第 53 行)、引用 (第 54 行) 和值 (第 55 行) 将该 Dog 对象分别传递给 3 个函数。

接受指针和引用的函数都调用虚方法, 因此调用成员函数 Dog->Speak(), 如第 2 行和第 3 行输出所示。

然而, 第 55 行解除指针引用, 并按值将结果传递给第 60~63 行定义的函数。函数希望接受一个 Mammal 对象, 因此编译器将 Dog 对象切除到只余下 Mammal 部分。第 62 行调用 Speak() 方法时, 只有 Mammal 的信息可用, Dog 部分没有了, 第 4 行输出表明了这一点。这种效果被称为切除, 因为转换为 Mammal (基类对象) 时, 对象的 Dog 部分 (派生类部分) 被切除了。

接下来, 对 Cat 对象重复这样的过程, 结果类似。

## 11.5.4 创建虚析构造函数

在需要基类指针的地方使用指向派生类对象的指针是一种合法和常见的做法。当指向派生对象的指针被删除时将发生什么情况呢? 如果析构函数是虚函数 (应该如此), 将执行正确的操作: 调用派生类的析构函数。由于派生类的析构函数会自动调用基类的析构函数, 因此整个对象将被正确地销毁。

经验规则是, 如果类中任何一个函数是虚函数, 析构函数也应该是虚函数。

### 注意

读者可能注意到了, 本章的程序清单中都包含虚析构函数。现在读者知道其原因了! 通常, 总是将析构函数声明为虚函数是明智的选择。

## 11.5.5 虚复制构造函数

构造函数不能是虚函数, 因此从技术上说, 不存在虚复制构造函数。然而, 有时候程序非常需要

通过传递一个指向基类对象的指针，创建一个派生类对象的副本。对于这种问题，一个常见的解决方法是，在基类种创建一个 Clone() 方法，并将其设置为虚方法。Clone() 方法创建当前类对象的一个副本，并返回该副本。

由于每个派生类都覆盖了 Clone() 方法，因此它将创建派生类对象的一个副本。程序清单 11.11 演示了如何使用 Clone() 方法。

程序清单 11.11 虚复制构造函数

```

1: //Listing 11.11 Virtual copy constructor
2: #include <iostream>
3: using namespace std;
4:
5: class Mammal
6: {
7:     public:
8:         Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
9:         virtual ~Mammal() { cout << "Mammal destructor...\n"; }
10:        Mammal (const Mammal & rhs);
11:        virtual void Speak() const { cout << "Mammal speak!\n"; }
12:        virtual Mammal* Clone() { return new Mammal(*this); }
13:        int GetAge()const { return itsAge; }
14:    protected:
15:        int itsAge;
16: };
17:
18: Mammal::Mammal (const Mammal & rhs):itsAge(rhs.GetAge())
19: {
20:     cout << "Mammal Copy Constructor...\n";
21: }
22:
23: class Dog : public Mammal
24: {
25:     public:
26:         Dog() { cout << "Dog constructor...\n"; }
27:         virtual ~Dog() { cout << "Dog destructor...\n"; }
28:         Dog (const Dog & rhs);
29:         void Speak()const { cout << "Woof!\n"; }
30:         virtual Mammal* Clone() { return new Dog(*this); }
31: };
32:
33: Dog::Dog(const Dog & rhs):
34:     Mammal(rhs)
35: {
36:     cout << "Dog copy constructor...\n";
37: }
38:
39: class Cat : public Mammal
40: {
41:     public:
42:         Cat() { cout << "Cat constructor...\n"; }
43:         ~Cat() { cout << "Cat destructor...\n"; }
44:         Cat (const Cat &);
45:         void Speak()const { cout << "Meow!\n"; }
46:         virtual Mammal* Clone() { return new Cat(*this); }
47: };
48:
49: Cat::Cat(const Cat & rhs):
50:     Mammal(rhs)
51: {
52:     cout << "Cat copy constructor...\n";
53: }
54:
55: enum ANIMALS { MAMMAL, DOG, CAT};
56: const int NumAnimalTypes = 3;
57: int main()
58: {
59:     Mammal *theArray[NumAnimalTypes];
60:     Mammal* ptr;
61:     int choice, i;

```



```
62:     for ( i = 0; i<NumAnimalTypes; i++)
63:     {
64:         cout << "(1)dog (2)cat (3)Mammal: ";
65:         cin >> choice;
66:         switch (choice)
67:         {
68:             case DOG: ptr = new Dog;
69:                     break;
70:             case CAT: ptr = new Cat;
71:                     break;
72:             default: ptr = new Mammal;
73:                     break;
74:         }
75:         theArray[i] = ptr;
76:     }
77:     Mammal *OtherArray[NumAnimalTypes];
78:     for (i=0;i<NumAnimalTypes;i++)
79:     {
80:         theArray[i]->Speak();
81:         OtherArray[i] = theArray[i]->Clone();
82:     }
83:     for (i=0;i<NumAnimalTypes;i++)
84:         OtherArray[i]->Speak();
85:     return 0;
86: }
```

#### ▼ 输出:

```
1: (1)dog (2)cat (3)Mammal: 1
2: Mammal constructor...
3: Dog constructor...
4: (1)dog (2)cat (3)Mammal: 2
5: Mammal constructor...
6: Cat constructor...
7: (1)dog (2)cat (3)Mammal: 3
8: Mammal constructor...
9: Woof!
10: Mammal Copy Constructor...
11: Dog copy constructor...
12: Meow!
13: Mammal Copy Constructor...
14: Cat copy constructor...
15: Mammal speak!
16: Mammal Copy Constructor...
17: Woof!
18: Meow!
19: Mammal speak!
```

#### ▼ 分析:

除第 12 行给 Mammal 类新增了虚方法 Clone( )外,程序清单 11.11 与前两个程序清单非常相似。这个方法通过调用复制构造函数,并将当前对象 (\*this) 作为 const 引用传递给它,返回了一个指向新 Mammal 对象的指针。

Dog 和 Cat 都覆盖了 Clone( )方法,使之调用相应类的复制构造函数,并将当前对象传递给它。由于 Clone( )是虚方法,因此这相当于创建了一个虚复制构造函数。当第 81 行执行时,读者将明白这一点。

和前一个程序类似,该程序也提示用户选择创建狗、猫或哺乳动物,然后第 68~73 行创建用户指定的对象。第 75 行将指向对象的指针存放到数组中。

程序在第 78~82 行遍历数组,并调用每个对象的方法 Speak( )和 Clone( )。第 81 行调用 Clone( )的结果为一个指向对象副本的指针,该指针被存储到另一个数组中。

第 1 行输出表明,用户被提示时选择了 1,即创建一个 Dog 对象,这将调用 Mammal 和 Dog 的构造函数。第 4 行和第 7 行输出表明,对 Cat 和 Mammal 重复了这个过程。

第 9 行输出为对第一个对象 (Dog 对象) 调用 Speak( )的结果。由于 Speak( )是一个虚方法,因此将调用正确的 Speak( )版本。接下来调用 Clone( )函数,由于它也是一个虚方法,因此调用 Dog 的 Clone( )

方法，这导致 Mammal 的构造函数和 Dog 的复制构造函数被调用。  
第 12~14 行输出是对 Cat 重复上述过程的结果；第 15~16 行是对 Mammal 重复上述过程的结果。最后，第 83~84 行遍历新的数组，通过其中的每个指针调用 Speak() 方法，结果如第 17~19 行的输出所示。

11.5.6 使用虚方法的代价

由于包含虚方法的类必须维护一个 v-table，因此使用虚方法会带来一些开销。如果类很小，且打算从它派生出其他类，则根本没有必要使用虚方法。  
将任何方法声明为虚方法后，便付出了创建 v-table 的大部分代价（虽然每增加一个表项都会增加一些内存开销）。在这种情况下，应将析构函数声明为虚函数，并假设其他所有方法也可能是虚方法。应仔细琢磨那些非虚方法，确保理解它们为什么不是虚方法。

应该	不应该
打算从一个类派生出其他类时，应在这个类中使用虚方法。 在任何方法为虚方法时，应将析构函数声明为虚函数。	不要将构造函数声明为虚函数。 不要试图通过派生类对象访问基类的私有数据。

11.6 私有继承

在前面的示例中，派生类 Dog 是以公有方式从基类 Mammal 派生而来的：

```
class Dog : public Mammal
```

这种派生建立的是 is-a 关系，它强调 Dog 的行为类似于 Mammal。采用这种继承时，Dog 类可使用 Mammal 类的公有和保护成员，并覆盖其虚方法。然而，有时候程序员想利用现有的基类（即通过派生使用现有的功能），但 is-a 关系没有意义，甚至是一种糟糕的编程方式。在这种情况下，私有继承将派上用场。

11.6.1 使用私有继承

假设要重用 ElectriMotor 类来创建 Fan 类。Fan 不是一个 ElectriMotor，而只是使用了 ElectriMotor，因此在公有继承这种情形下不合适，但程序员可以这样做：

```
class Fan : private ElectricMotor
```

这种派生让 Fan 能够使用 ElectriMotor 的所有功能，甚至覆盖其虚函数。然而，Fan 的用户（如拥有 Fan 对象的人）不能访问基类 ElectriMotor 及其函数——对 Fan 的用户来说，该基类的所有内容都是私有的，而不管其使用的访问权限限定符是什么。程序清单 11.2 说明了私有继承导致的这些限制。

程序清单 11.12 私有继承

```
1: //Listing 11.12 Demonstration of Private Inheritance
2: #include <iostream>
3: using namespace std;
4:
5: class ElectricMotor
6: {
7: public:
8:     ElectricMotor () {} ;
9:     virtual ~ElectricMotor () {} ;
10:
11: public:
12:     void StartMotor ()
13:     {
```

```

14:     Accelerate ();
15:     Cruise ();
16: }
17:
18: void StopMotor ()
19: {
20:     cout << "Motor stopped" << endl;
21: };
22:
23: private:
24:     void Accelerate ()
25:     {
26:         cout << "Motor started" << endl;
27:     }
28:
29:     void Cruise ()
30:     {
31:         cout << "Motor running at constant speed" << endl;
32:     }
33: };
34:
35: class Fan : private ElectricMotor
36: {
37: public:
38:     Fan () {}
39:     ~Fan () {}
40:
41:     void StartFan ()
42:     {
43:         StartMotor ();
44:     }
45:
46:     void StopFan ()
47:     {
48:         StopMotor ();
49:     }
50: };
51:
52: int main ()
53: {
54:     Fan mFan;
55:
56:     mFan.StartFan ();
57:     mFan.StopFan ();
58:
59:     /*
60:         Note: the next two lines access the base class ElectricMotor
61:         However, as Fan features 'private inheritance' from ElectricMotor,
62:         neither the base class instance nor its public methods are
63:         accessible to the users of class Fan.
64:         Un-comment them to see a compile failure!
65:     */
66:     // mFan.Accelerate ();
67:     // ElectricMotor * pMotor = &mFan;
68:
69:     return 0;
70: }

```

### ▼ 分析:

上述代码很简单，它演示了通过私有继承让 Fan 类能够使用 ElectricMotor 的公有方法，同时禁止 Fan 对象的用户使用这些方法。第 66 行和第 67 行试图通过 mFan 访问基类 ElectricMotor 的实例，但它们不能通过派生类 Fan 的实例访问基类 ElectricMotor 的任何公有方法，因为这两个类之间的继承关系是私有的，因此禁止这种访问。如果解除对这两行的注释，将导致编译错误。

## 11.6.2 私有继承和聚合（组合）

可对程序清单 11.12 进行修改，将 ElectricMotor 作为 Fan 的私有成员，而不是以私有方式从

ElectricMotor 派生出 Fan。这称为聚合或组合。为理解这种概念,想象 Fan 类可包含一个 ElectricMotor 对象数组,即聚合了 ElectricMotor 类。程序清单 11.13 演示了这一点。

程序清单 11.13 聚合 ElectricMotor 的 Fan 类

```
1: //Listing 11.13 Version of class Fan that aggregates ElectricMotor
2: class Fan
3: {
4: public:
5:     Fan () {}
6:     ~Fan () {}
7:
8:     void StartFan ()
9:     {
10:         m_ElectricMotor.StartMotor();
11:     }
12:     void StopFan ()
13:     {
14:         m_ElectricMotor.StopMotor ();
15:     }
16: private:
17:     ElectricMotor m_ElectricMotor;
18: };
```

#### ▼ 分析:

这个版本的复杂程度与使用私有继承相当,但程序员可对这个版本的 Fan 类进行修改,使其包含一组 ElectricMotor 对象,即从只有一个 ElectricMotor 对象成员改为包含一个 ElectricMotor 数组。

与使用私有继承相比,使用聚合的另一个不那么明显的优点是,这消除了继承层次结构,从而避免了“使用虚方法的代价”以及讨论的 v-table,进而可提高性能。

私有继承的优点如下:

- 派生类能够访问基类的保护成员函数;
- 派生类可以覆盖基类的虚函数。

但与这些优点相伴而行的是如下代价:

- 在多程序员编程环境中,代码的灵活性更低,更容易出错;
- 继承带来的性能问题。

因此,在可以使用聚合或组合的情况下,最好避免使用私有继承。

## 11.7 总结

本章介绍了派生类如何继承基类,讨论了公有继承和虚函数。派生类将继承其基类的所有公有和保护数据和函数。

对派生类来说,基类的所有保护成员都是公有的,而对其他类来说是私有的。即使是派生类,也不能访问其基类的私有数据和函数。

可在派生类的构造函数开头初始化基类,这是通过调用基类的构造函数并给它传递参数完成的。

在派生类中,可以覆盖基类的函数。如果函数是虚函数,通过指针或引用来调用它时,将根据指针或引用在运行阶段指向的对象类型调用相应类的该函数。

通过加上由基类名和两个冒号组成的前缀,可以调用基类的函数。例如,如果 Dog 是从 Mammal 派生而来的,则可以使用 Mammal::Walk() 来调用 Mammal 的 Walk() 方法。

在包含虚方法的类中,几乎总是应该将析构函数声明为虚函数。虚析构函数确保将 delete 用于指针时,指针指向的对象的派生部分得到释放。构造函数不能是虚函数。创建一个调用复制构造函数的虚函数,相当于创建了一个虚复制构造函数。

## 11.8 问与答

问：继承而来的成员和函数能传给下一代吗？如果 Dog 从 Mammal 派生而来，而 Mammal 从 Animal 派生而来，Dog 将继承 Animal 的函数和数据吗？

答：可以。随着派生的进行，派生类将继承其所有基类的所有的函数和数据，但只能访问公有或保护函数和数据。

问：在上面的例子中，如果 Mammal 覆盖了 Animal 的一个函数，Dog 将继承原来的函数还是覆盖后的函数？

答：如果 Dog 从 Mammal 派生而来，将继承覆盖后的函数。

问：派生类可以把公有基类函数变成私有的吗？

答：可以。派生类可以覆盖该函数，使其变成私有的。这样，在接下来的派生类中，它将继续为私有的。然而，应尽可能避免这样做，因为用户期望类包含其祖先的所有方法。

问：为什么不将类函数都声明为虚函数？

答：创建 v-table 表的开销伴随第一个虚方法的创建而发生。在此之后，创建其他虚方法带来的开销将很小。很多 C++ 程序员认为，如果一个函数为虚函数，其他所有函数也应为虚函数；另一些程序员不同意这样观点，他们认为，无论做什么都应有充分的理由。

问：如果基类的一个函数（SomeFunc()）是虚函数，且被重载以便能接受一个或两个 int 参数，而派生类覆盖了接受一个 int 参数的版本，则通过指向派生类对象的指针调用接受两个 int 参数的函数时，将调用哪个函数？

答：正如本章指出的，覆盖接受一个 int 参数的版本将隐藏基类中所有与此同名的函数，因此将出现编译错误，指出该函数只接受一个 int 参数。

## 11.9 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 11.9.1 测验

1. 什么是虚函数表？
2. 什么是虚析构函数？
3. 如何声明虚构造函数？
4. 如何创建一个虚复制构造函数？
5. 如何在派生类调用被覆盖的基类成员函数？
6. 如何在派生类调用未被覆盖的基类成员函数？
7. 如果基类将一个函数声明为虚函数，而派生类在覆盖这个函数时没有使用关键字 virtual，则被第 3 代类继承时，该函数是否仍为虚函数？
8. 关键字 protected 的作用是什么？



### 11.9.2 练习

1. 声明一个接受一个 int 参数且返回 void 的虚函数。
2. 声明 Square 类，它从 Rectangle 派生而来，而 Rectangle 又从 Shape 派生而来。
3. 如果在练习 2 中，Shape 的构造函数不接受任何参数，Rectangle 的构造函数接受两个参数（长度和宽度），而 Square 的构造函数接受一个参数（长度），请写出 Square 的构造函数的初始化部分。
4. 为练习 3 中的 Square 类编写一个虚复制构造函数。
5. 查错：下面这段代码有什么错误？

```
void SomeFunction (Shape);  
Shape * pRect = new Rectangle;  
SomeFunction(*pRect);
```

6. 查错：下面这段代码有什么错误？

```
class Shape()  
{  
    public:  
        Shape();  
        virtual ~Shape();  
        virtual Shape(const Shape&);  
};
```



# 第 12 章

## 多 态

第 11 章介绍了如何在派生类中编写虚函数，这是多态的基石：在运行阶段将派生类对象绑定到基类指针。

在本章中，您将学习：

- 什么是多重继承及如何使用它
- 什么是虚继承及何时使用它
- 什么是抽象类及何时使用它们
- 什么是纯虚函数

### 12.1 单继承存在的问题

假设您使用动物类已有一段时间了，且将类层次结构分为了鸟类和哺乳动物。Bird 类包括成员函数 Fly()。从 Mammal 类派生出了包括 Horse 在内的很多哺乳动物类，Horse 类包括成员函数 Whinny() 和 Gallop()。

您突然意识到您需要一个 Pegasus（飞马）对象：一种介于马和鸟之间的动物。Pegasus 类可包含成员函数 Fly()、Whinny() 和 Gallop()。如果使用单继承，您将陷入困境。

使用单继承时，只能从一个类派生而来。可以从 Bird 派生出 Pegasus，但这样它将不会有成员函数 Whinny() 和 Gallop()；也可从 Horse 派生出 Pegasus，但这样它将不会有成员函数 Fly()。

第一种解决方法是，从 Horse 类派生出 Pegasus，并将 Fly() 方法复制到 Pegasus 类中。这是可行的，但代价是 Fly() 方法位于两个地方（Bird 和 Pegasus 类中）。如果修改了其中一个，必须修改另一个。当然，数月或数年后负责维护代码的开发人员也必须知道需要修改这两个地方。

然而，很快又出现了新问题。您想创建一个 Horse 对象链表和一个 Bird 对象链表。您希望可以将 Pegasus 对象添加到任何一个链表中，但如果 Pegasus 是从 Horse 派生而来的，将不能把它添加到 Birds 链表中。

有两种可能的解决方案。可以将 Horse 的方法 Gallop() 重命名为 Move()，然后在 Pegasus 类中覆盖 Move() 使其完成 Fly() 的工作，再在其他从 Horse 派生而来的类中覆盖 Move() 使其完成 Gallop() 的工作。也许 Pegasus 足够聪明，能够奔跑较短的距离和飞翔更长的距离，如下例所示：

```
Pegasus::Move(long distance)
{
    if (distance > veryFar)
        fly(distance);
    else
        gallop(distance);
}
```

这段代码有一定的局限性。也许有朝一日，Pegasus 想飞翔较短的距离或奔跑较长的距离。另一种解决方法是，将 Fly() 方法移到 Horse 类中，如程序清单 12.1 所示。问题是大多数马都不能飞，因此必须使这个方法什么也不做，除非其所属类为 Pegasus。

## 程序清单 12.1 如果马能飞

```

0: // Listing 12.1. If horses could fly...
1: // Percolating Fly() up into Horse
2:
3: #include <iostream>
4: using namespace std;
5:
6: class Horse
7: {
8:     public:
9:         void Gallop(){ cout << "Galloping...\n"; }
10:        virtual void Fly() { cout << "Horses can't fly.\n" ; }
11:    private:
12:        int itsAge;
13: };
14:
15: class Pegasus : public Horse
16: {
17:     public:
18:         void Fly()
19:             {cout<<"I can fly! I can fly! I can fly!\n";}
20: };
21:
22: const int NumberHorses = 5;
23: int main()
24: {
25:     Horse* Ranch[NumberHorses];
26:     Horse* pHorse;
27:     int choice,i;
28:     for (i=0; i<NumberHorses; i++)
29:     {
30:         cout << "(1)Horse (2)Pegasus: ";
31:         cin >> choice;
32:         if (choice == 2)
33:             pHorse = new Pegasus;
34:         else
35:             pHorse = new Horse;
36:         Ranch[i] = pHorse;
37:     }
38:     cout << endl;
39:     for (i=0; i<NumberHorses; i++)
40:     {
41:         Ranch[i]->Fly();
42:         delete Ranch[i];
43:     }
44:     return 0;
45: }

```

## ▼ 输出:

```

(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1

```

```

Horses can't fly.
I can fly! I can fly! I can fly!
Horses can't fly.
I can fly! I can fly! I can fly!
Horses can't fly.

```

## ▼ 分析:

该程序肯定可行, 虽然其代价是 Horse 类有一个 Fly() 方法。第 10 行为 Horse 类提供了 Fly() 方法。在现实世界的类中, 您可能让它发出错误信号或平静地失败。在第 18 行, Pegasus 类覆盖了 Fly() 方法使其“完成正确的工作”, 这里通过打印一条快乐消息来表示。

第 25 行使用名为 Ranch 的 Horse 指针数组来演示这一点: 将根据运行阶段绑定的 Horse 或 Pegasus

对象正确地调用 Fly() 方法。

第 28~37 行提示用户选择 Horse 或 Pegasus, 然后根据用户的选择创建相应类型的对象, 并将其放到数组 Ranch 中。

在第 39~43 行, 程序通过循环来遍历数组 Ranch, 对数组中每个对象调用 Fly() 方法。将根据对象是 Horse 还是 Pegasus 调用正确的 Fly() 方法, 输出证明了这一点。由于程序不再使用数组 Ranch 中的对象, 因此第 42 行调用 delete 来释放每个对象占用的内存。

#### 注意

为突出要讨论的重点, 对这些例子做了删节, 只保留最基本的内容。为简化代码, 删去了构造函数、虚析构函数等。在实际的程序中不推荐这样做。

### 12.1.1 提升

将所需的函数放到类层次结构中较高的位置, 是一种常用的解决这种问题的方法, 其结果是很多函数被提升 (percolating up) 到基类中。这样, 基类很可能变成派生类可能使用的所有函数的全局名称空间。这将严重破坏 C++ 类的类型化 (class typing), 创建出庞杂的基类。

一般而言, 将共有的功能沿层次结构向上提升, 而不用迁移每个类的接口。这意味着如果两个类有相同的基类 (如 Horse 和 Bird 的基类都是 Animal) 且有一项相同的功能 (如鸟和马都会吃), 则应将该功能移到基类中, 在其中创建一个虚函数。

然而需要避免的是, 仅仅为了能够在某些派生类中调用函数而将它 (如 Fly()) 提升到不属于它的地方 (不符合基类的含义)。

### 12.1.2 向下转换

采用单继承时, 另一种办法是将把 Fly() 方法留在 Pegasus 中, 且仅当指针指向 Pegasus 对象时才调用它。为此, 需要知道指针实际指向的类型, 这被称为运行阶段类型识别 (Runtime Type Identification, RTTI)。

#### 警告

在程序中应慎用 RTTI。需要使用 RTTI 可能意味着继承层次结构设计得很糟糕。应考虑使用虚函数、模板或多重继承来代替它。

在这个例子中, 声明了 Horse 和 Pegasus 对象并将其存储到一个 Horse 对象数组中, 每个对象都是作为 Horse 对象存储到数组中的。使用 RTTI, 可检查每个对象看它实际上是 Horse 还是 Pegasus。

然而为调用 Fly(), 必须对指针进行类型转换以告诉指针, 它指向的对象是 Pegasus 而不是 Horse。这被称为向下转换 (casting down), 因为您将 Horse 对象向下转换为派生类型。C++ 使用 dynamic\_cast 运算符支持向下转换, 其工作原理如下。

如果有一个指向基类 (如 Horse) 的指针, 并将指向派生类 (如 Pegasus) 的指针赋给它, 便可以多态方式使用 Horse 指针。如果需要访问 Pegasus 对象, 可创建一个 Pegasus 指针并使用 dynamic\_cast 运算符进行转换。

在运行阶段, 将检查基类指针。如果转换正确, 新的 Pegasus 指针也是正确的, 如果转换不正确或根本没有 Pegasus 对象, 新指针将为空。程序清单 12.2 演示了这一点。

程序清单 12.2 使用 RTTI 向下转换

```
0: // Listing 12.2 Using dynamic_cast.
1: // Using rtti
2:
3: #include <iostream>
4: using namespace std;
5:
6: enum TYPE { HORSE, PEGASUS };
7:
8: class Horse
```

```

9: {
10:     public:
11:         virtual void Gallop(){ cout << "Galloping...\n"; }
12:
13:     private:
14:         int itsAge;
15: };
16:
17: class Pegasus : public Horse
18: {
19:     public:
20:         virtual void Fly()
21:             {cout<<"I can fly! I can fly! I can fly!\n";}
22: };
23:
24: const int NumberHorses = 5;
25: int main()
26: {
27:     Horse* Ranch[NumberHorses];
28:     Horse* pHorse;
29:     int choice,i;
30:     for (i=0; i<NumberHorses; i++)
31:     {
32:         cout << "(1)Horse (2)Pegasus: ";
33:         cin >> choice;
34:         if (choice == 2)
35:             pHorse = new Pegasus;
36:         else
37:             pHorse = new Horse;
38:         Ranch[i] = pHorse;
39:     }
40:     cout << endl;
41:     for (i=0; i<NumberHorses; i++)
42:     {
43:         Pegasus *pPeg = dynamic_cast< Pegasus *> (Ranch[i]);
44:         if (pPeg != NULL)
45:             pPeg->Fly();
46:         else
47:             cout << "Just a horse\n";
48:         delete Ranch[i];
49:     }
50:     return 0;
51: }
52: }

```

### ▼ 输出:

```

(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1

```

```

Just a horse
I can fly! I can fly! I can fly!
Just a horse
I can fly! I can fly! I can fly!
Just a horse

```

### FAQ

使用微软 Visual C++ 进行编译时出现如下警告:

```
warning C4541: 'dynamic_cast' used on polymorphic type 'class
Horse' with /GR-; unpredictable behavior may result.
```

我该怎么办? 运行该程序时, 出现如下消息:

```
This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

答: 这是该编译器最让人迷惑的错误消息之一, 执行如下步骤可解决这个问题。



1. 在项目中选择菜单 Project/Settings.

2. 打开 C++选项卡.

3. 从下拉式列表中选择 C++ Language.

4. 选中复选框 Enable Runtime Type Information(RTTI).

5. 重新编译整个项目.

▼ 分析:

这种解决方案也可行，但不推荐这样做。

这取得预期的效果。Fly()方法不在 Horse 中，没有对 Horse 对象调用它。然而，对 Pegasus 对象中调用 Fly()时（第 45 行），必须进行显式转换（第 43 行）。由于 Horse 对象没有方法 Fly()，因此必须在使用指针前告诉它，它指向的是 Pegasus 对象。

需要对 Pegasus 对象进行转换意味着设计可能有问题。该程序实际上破坏了虚函数的多态性，因为它依赖于将对象转换为运行阶段类型。

12.1.3 将对象添加到链表中

这些解决方案存在的另一个问题是，Pegasus 被声明为一种 Horse，所以不能将 Pegasus 对象添加到 Birds 链表中。您付出了将 Fly()方法移到 Horse 中或对指针进行向下转换的代价，却没有获得所需的全部功能。

最后一种单继承解决方案是，将 Fly()、Whinny()和 Gallop()都提升到 Bird 和 Horse 的基类 Animal 中。现在，可以使用一个统一的 Animals 链表，而不是分别使用一个 Birds 链表和一个 Horse 链表。这种解决方案也可行，但导致将派生类的所有特征都放在基类中。既然如此，谁还要派生类呢？

另一种方法是将这些方法留在原来的地方，对 Horses、Birds 和 Pegasus 对象进行向下转换，但这样做更糟。

应该	不应该
<div>务必将在概念上符合基类含义的功能沿继承层次向上提升。</div> <div>务必避免根据对象的运行阶段类型执行操作，而应使用虚函数、模板和多重继承。</div>	<div>不要将为支持多态而本应在派生类中添加的功能放到基类中。</div> <div>不要将基类指针向下转换为派生类指针。</div>

12.2 多重继承

可以从多个基类派生出新类，这被称为多重继承。要从多个基类派生，在类声明中将每个基类用逗号分开，如下所示：

```
class DerivedClass : public BaseClass1, public BaseClass2 {}
```

除添加了基类 BaseClass2 外，这与声明单继承完全相同。

程序清单 12.3 说明了如何声明 Pegasus，使其从 Horse 和 Birds 派生而来。该程序然后将 Pegasus 对象添加到两种类型的链表中。

程序清单 12.3 多重继承

```
0: // Listing 12.3. Multiple inheritance.
1:
2: #include <iostream>
3: using std::cout;
```

```
4: using std::cin;
5: using std::endl;
6:
7: class Horse
8: {
9:     public:
10:    Horse() { cout << "Horse constructor... "; }
11:    virtual ~Horse() { cout << "Horse destructor... "; }
12:    virtual void Whinny() const { cout << "Whinny!... "; }
13:    private:
14:    int itsAge;
15: };
16:
17: class Bird
18: {
19:     public:
20:    Bird() { cout << "Bird constructor... "; }
21:    virtual ~Bird() { cout << "Bird destructor... "; }
22:    virtual void Chirp() const { cout << "Chirp... "; }
23:    virtual void Fly() const
24:    {
25:        cout << "I can fly! I can fly! I can fly! ";
26:    }
27:    private:
28:    int itsWeight;
29: };
30:
31: class Pegasus : public Horse, public Bird
32: {
33:     public:
34:    void Chirp() const { Whinny(); }
35:    Pegasus() { cout << "Pegasus constructor... "; }
36:    ~Pegasus() { cout << "Pegasus destructor... "; }
37: };
38:
39: const int MagicNumber = 2;
40: int main()
41: {
42:     Horse* Ranch[MagicNumber];
43:     Bird* Aviary[MagicNumber];
44:     Horse * pHorse;
45:     Bird * pBird;
46:     int choice,i;
47:     for (i=0; i<MagicNumber; i++)
48:     {
49:         cout << "\n(1)Horse (2)Pegasus: ";
50:         cin >> choice;
51:         if (choice == 2)
52:             pHorse = new Pegasus;
53:         else
54:             pHorse = new Horse;
55:         Ranch[i] = pHorse;
56:     }
57:     for (i=0; i<MagicNumber; i++)
58:     {
59:         cout << "\n(1)Bird (2)Pegasus: ";
60:         cin >> choice;
61:         if (choice == 2)
62:             pBird = new Pegasus;
63:         else
64:             pBird = new Bird;
65:         Aviary[i] = pBird;
66:     }
67:
68:     cout << endl;
69:     for (i=0; i<MagicNumber; i++)
70:     {
71:         cout << "\nRanch[" << i << "]: ";
72:         Ranch[i]->Whinny();
73:         delete Ranch[i];
74:     }
75:
76:     for (i=0; i<MagicNumber; i++)
```

```

77:    {
78:        cout << "\nAviary[" << i << "]: " ;
79:        Aviary[i]->Chirp();
80:        Aviary[i]->Fly();
81:        delete Aviary[i];
82:    }
83:    return 0;
84: }

```

### ▼ 输出:

```

(1)Horse (2)Pegasus: 1
Horse constructor...
(1)Horse (2)Pegasus: 2
Horse constructor... Bird constructor... Pegasus constructor...
(1)Bird (2)Pegasus: 1
Bird constructor...
(1)Bird (2)Pegasus: 2
Horse constructor... Bird constructor... Pegasus constructor...

Ranch[0]: Whinny!... Horse destructor...
Ranch[1]: Whinny!... Pegasus destructor... Bird destructor...
Horse destructor...
Aviary[0]: Chirp... I can fly! I can fly! I can fly! Bird destructor...
Aviary[1]: Whinny!... I can fly! I can fly! I can fly!
Pegasus destructor... Bird destructor... Horse destructor...

```

### ▼ 分析:

第 7~15 行声明了 Horse 类，其构造函数和析构函数分别打印一条消息，而 Whinny() 方法打印“Whinny!....”。

第 17~29 行声明了 Birds 类。除构造函数和析构函数外，这个类还有两个方法：Chirp() 和 Fly()，它们都打印识别消息。在实际程序中，它们可能激活扬声器或生成动画图像。

最后，第 31~37 行是新的代码，使用多重继承声明了 Pegasus 类。从第 31 行可知，这个类是从 Horse 和 Bird 派生而来的。在第 34 行，Pegasus 类覆盖了 Chirp() 方法。Pegrsus 的 Chrip() 只是调用从 Horse 类继承的 Whinny() 方法。

在该程序的 main() 函数中创建了两个链表：第 42 行创建了一个名为 Ranch 的 Horse 指针数组，第 43 行创建了名为 Aviary 的 Bird 指针数组。第 47~56 行将 Horse 和 Pegasus 对象添加到数组 Ranch 中，第 57~66 行将 Bird 和 Pegasus 对象添加到数组 Aviary 中。

通过 Bird 指针和 Horse 指针调用虚方法时，都会为 Pegasus 对象执行正确的操作。例如，第 79 行使用数组 Aviary 的元素来对其指向的对象调用 Chirp()。Bird 类将该方法声明为虚方法，因此对每个对象都调用了正确的函数。

输出表明，每次创建 Pegasus 对象时都创建了其 Bird 部分和 Horse 部分。当 Pegasus 对象被销毁时，其 Bird 和 Horse 部分也被销毁，这要归功于析构函数被声明为虚函数。

#### 声明多重继承

要声明从多个类派生而来的类，在类名后加上一个冒号，再加上基类名即可。基类名之间用逗号分开。

示例 1:

```
class Pegasus : public Horse, public Bird
```

示例 2:

```
class Schnoodle : public Schnauzer, public Poodle
```

## 12.2.1 多重继承对象的组成部分

在内存中创建 Pegasus 对象时，两个基类都将成为 Pegasus 对象的组成部分，如图 12.1 所示。图中

表示的是整个 Pegasus 对象，这包括 Pegasus 类新增的功能以及从基类那里继承而来的功能。

有多个基类的对象存在一些问题。例如，如果碰巧两个基类有同名的虚函数或数据将如何呢？如何调用多个基类构造函数？如果多个基类都从同一个类派生而来又将如何呢？接下来的几节将回答这些问题并探讨如何使多重继承可行。

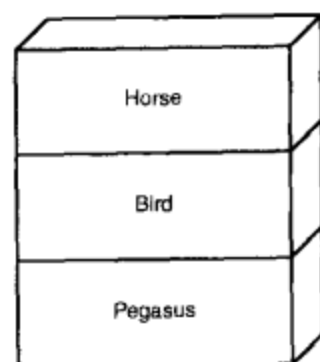


图 12.1 多重继承对象

### 12.2.2 多重继承对象中的构造函数

如果 Pegasus 从 Horse 和 Bird 类派生而来且每个基类都有接受参数的构造函数，Pegasus 将依次调用这些构造函数。程序清单 12.4 说明了这是如何完成的。

程序清单 12.4 使用重载的基类构造函数

```

0: // Listing 12.4
1: // Calling multiple constructors
2:
3: #include <iostream>
4: using namespace std;
5:
6: typedef int HANDS;
7: enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };
8:
9: class Horse
10: {
11: public:
12:     Horse(COLOR color, HANDS height);
13:     virtual ~Horse() { cout << "Horse destructor...\n"; }
14:     virtual void Whinny()const { cout << "Whinny!... "; }
15:     virtual HANDS GetHeight() const { return itsHeight; }
16:     virtual COLOR GetColor() const { return itsColor; }
17: private:
18:     HANDS itsHeight;
19:     COLOR itsColor;
20: };
21:
22: Horse::Horse(COLOR color, HANDS height):
23:     itsColor(color), itsHeight(height)
24: {
25:     cout << "Horse constructor...\n";
26: }
27:
28: class Bird
29: {
30: public:
31:     Bird(COLOR color, bool migrates);
32:     virtual ~Bird() {cout << "Bird destructor...\n"; }
33:     virtual void Chirp()const { cout << "Chirp... "; }
34:     virtual void Fly()const
35:     {
36:         cout << "I can fly! I can fly! I can fly! ";
37:     }
38:     virtual COLOR GetColor()const { return itsColor; }
39:     virtual bool GetMigration() const { return itsMigration; }
40: private:
41:     COLOR itsColor;
42:     bool itsMigration;
43: };
44:
45: Bird::Bird(COLOR color, bool migrates):
46:     itsColor(color), itsMigration(migrates)
47: {
48:     cout << "Bird constructor...\n";
49: }
50:

```

```
51:
52: class Pegasus : public Horse, public Bird
53: {
54:     public:
55:         void Chirp()const { Whinny(); }
56:         Pegasus(COLOR, HANDS, bool, long);
57:         ~Pegasus() {cout << "Pegasus destructor...\n";}
58:         virtual long GetNumberBelievers() const
59:         {
60:             return itsNumberBelievers;
61:         }
62:
63:     private:
64:         long itsNumberBelievers;
65: };
66:
67: Pegasus::Pegasus(
68:     COLOR aColor,
69:     HANDS height,
70:     bool migrates,
71:     long NumBelieve):
72:     Horse(aColor, height),
73:     Bird(aColor, migrates),
74:     itsNumberBelievers(NumBelieve)
75: {
76:     cout << "Pegasus constructor...\n";
77: }
78:
79: int main()
80: {
81:     Pegasus *pPeg = new Pegasus(Red, 5, true, 10);
82:     pPeg->Fly();
83:     pPeg->Whinny();
84:     cout << "\nYour Pegasus is " << pPeg->GetHeight();
85:     cout << " hands tall and ";
86:     if (pPeg->GetMigration())
87:         cout << "it does migrate.";
88:     else
89:         cout << "it does not migrate.";
90:     cout << "\nA total of " << pPeg->GetNumberBelievers();
91:     cout << " people believe it exists." << endl;
92:     delete pPeg;
93:     return 0;
94: }
```

### ▼ 输出:

```
Horse constructor...
Bird constructor...
Pegasus constructor...
I can fly! I can fly! I can fly! Whinny!...
Your Pegasus is 5 hands tall and it does migrate.
A total of 10 people believe it exists.
Pegasus destructor...
Bird destructor...
Horse destructor...
```

### ▼ 分析:

第 9~20 行声明了 Horse 类，其构造函数接受两个参数：一个是第 7 行声明的颜色枚举类型，另一个是第 6 行声明的 typedef。第 22~26 行的构造函数实现只是初始化成员变量并打印一条消息。

第 28~44 行声明了 Bird 类，其构造函数的实现位于第 46~50 行。Bird 类的构造函数也接受两个参数。有趣的是，Horse 构造函数将颜色作为参数（以便您可以识别不同颜色的马），而 Bird 构造函数将羽毛颜色作为参数（同一种羽毛的鸟可友好相处）。当您想了解 Pegasus 的颜色时将引发一个问题，这将在程序清单 12.5 中看到。

Pegasus 类的声明位于第 52~65 行，其构造函数是在第 67~77 实现的。Pegasus 对象的初始化部分包括 3 条语句：首先，用颜色和高度作为参数调用 Horse 构造函数（第 72 行），然后，用颜色和指



出是否将迁徙的 Boolean 值作为参数调用 Bird 构造函数（第 73 行）；最后，初始化 Pegasus 的成员变量 ItsNumberBelievers。完成这些工作后，执行 Pegasus 的构造函数体。

在 main() 函数中，第 81 行创建了一个 Pegasus 指针，然后使用它来访问从基类那里继承而来的成员函数。对这些方法的访问非常简单。

### 12.2.3 避免歧义

在程序清单 12.4 中，Horse 类和 Bird 类都有方法 GetColor()。读者可能注意到了，在程序清单 12.4 中没有调用这些方法！您可能需要让 Pegasus 对象返回其颜色，但将遇到问题：Pegasus 是从 Bird 和 Horse 类派生而来的，它们都有颜色，且获得颜色的方法的名称和特征标相同。这将给编译器带来歧义，您必须澄清。

如果编写下面这样的代码：

```
COLOR currentColor = pPeg->GetColor();
```

您出现这样的编译错误：

```
Member is ambiguous: 'Horse::GetColor' and 'Bird::GetColor'
```

显式地调用要调用的函数可解决这种歧义问题：

```
COLOR currentColor = pPeg->Horse::GetColor();
```

要指出从哪个类继承而来的成员函数或成员数据时，都可以通过在数据或函数前加上基类名来全限定调用。

注意，如果 Pegasus 覆盖了这个函数，这个问题将挪到 Pegasus 成员函数中去（本该如此）：

```
virtual COLOR GetColor()const { return Horse::GetColor(); }
```

这样就对 Pegasus 类的客户隐藏了问题，并在 Pegasus 中封装了它要从哪个基类继承颜色的知识。

客户仍可通过编写下面的代码来解决这种问题：

```
COLOR currentColor = pPeg->Bird::GetColor();
```

### 12.2.4 从共同基类继承

如果 Bird 和 Horse 是从同一个基类（如 Animal）派生而来的，情况将如何呢？图 12.2 说明了这种情形。

从图 12.2 可知，有两个基类对象。调用共同基类的函数或数据成员时，将出现另一种歧义。例如，如果 Animal 将 itsAge 声明为其成员变量，将 GetAge() 声明为其成员函数，当您调用 pPeg->GetAge() 时，指的是通过 Horse 还是 Bird 从 Animal 那里继承而来的 GetAge() 呢？必须解决这种歧义问题，如程序清单 12.5 所示。

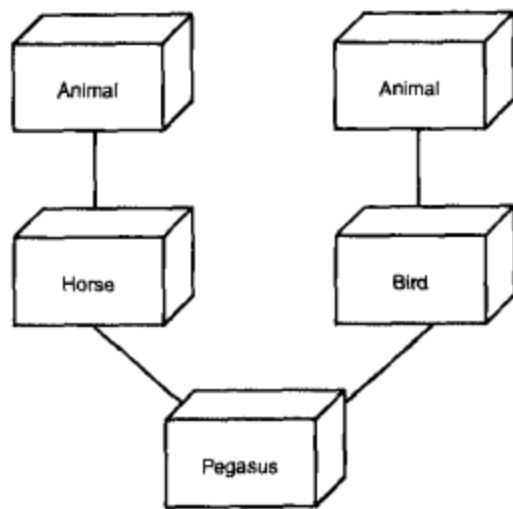


图 12.2 相同的基类

程序清单 12.5 在有共同基类的情况下避免多重继承的歧义

```
0: // Listing 12.5
1: // Common base classes
2:
3: #include <iostream>
4: using namespace std;
5:
6: typedef int HANDS;
7: enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };
8:
9: class Animal // common base to both horse and bird
10: {
11:     public:
12:         Animal(int);
```

```

13:     virtual ~Animal() { cout << "Animal destructor...\n"; }
14:     virtual int GetAge() const { return itsAge; }
15:     virtual void SetAge(int age) { itsAge = age; }
16: private:
17:     int itsAge;
18: };
19:
20: Animal::Animal(int age):
21: itsAge(age)
22: {
23:     cout << "Animal constructor...\n";
24: }
25:
26: class Horse : public Animal
27: {
28: public:
29:     Horse(COLOR color, HANDS height, int age);
30:     virtual ~Horse() { cout << "Horse destructor...\n"; }
31:     virtual void Whinny()const { cout << "Whinny!... "; }
32:     virtual HANDS GetHeight() const { return itsHeight; }
33:     virtual COLOR GetColor() const { return itsColor; }
34: protected:
35:     HANDS itsHeight;
36:     COLOR itsColor;
37: };
38:
39: Horse::Horse(COLOR color, HANDS height, int age):
40: Animal(age),
41: itsColor(color),itsHeight(height)
42: {
43:     cout << "Horse constructor...\n";
44: }
45:
46: class Bird : public Animal
47: {
48: public:
49:     Bird(COLOR color, bool migrates, int age);
50:     virtual ~Bird() {cout << "Bird destructor...\n"; }
51:     virtual void Chirp()const { cout << "Chirp... "; }
52:     virtual void Fly()const
53:         { cout << "I can fly! I can fly! I can fly! "; }
54:     virtual COLOR GetColor()const { return itsColor; }
55:     virtual bool GetMigration() const { return itsMigration; }
56: protected:
57:     COLOR itsColor;
58:     bool itsMigration;
59: };
60:
61: Bird::Bird(COLOR color, bool migrates, int age):
62: Animal(age),
63: itsColor(color), itsMigration(migrates)
64: {
65:     cout << "Bird constructor...\n";
66: }
67:
68: class Pegasus : public Horse, public Bird
69: {
70: public:
71:     void Chirp()const { Whinny(); }
72:     Pegasus(COLOR, HANDS, bool, long, int);
73:     virtual ~Pegasus() {cout << "Pegasus destructor...\n";}
74:     virtual long GetNumberBelievers() const
75:         { return itsNumberBelievers; }
76:     virtual COLOR GetColor()const { return Horse::itsColor; }
77:     virtual int GetAge() const { return Horse::GetAge(); }
78: private:
79:     long itsNumberBelievers;
80: };
81:
82: Pegasus::Pegasus(
83:     COLOR aColor,
84:     HANDS height,
85:     bool migrates,

```

```
86:     long NumBelieve,
87:     int age):
88:     Horse(aColor, height, age),
89:     Bird(aColor, migrates, age),
90:     itsNumberBelievers(NumBelieve)
91: {
92:     cout << "Pegasus constructor...\n";
93: }
94:
95: int main()
96: {
97:     Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
98:     int age = pPeg->GetAge();
99:     cout << "This pegasus is " << age << " years old.\n";
100:    delete pPeg;
101:    return 0;
102: }
```

#### ▼ 输出:

```
Animal constructor...
Horse constructor...
Animal constructor...
Bird constructor...
Pegasus constructor...
This pegasus is 2 years old.
Pegasus destructor...
Bird destructor...
Animal destructor...
Horse destructor...
Animal destructor...
```

#### ▼ 分析:

该程序清单有多个有趣的特性。Animal 类的声明位于第 9~18 行,它新增了一个成员变量 (itsAge) 和两个存取器函数 (GetAge() 和 SetAge())。

第 26 行将 Horse 声明为从 Animal 类派生而来。现在 Horse 构造函数接受第 3 个参数 (年龄), 它将该参数传递给基类 Animal (第 40 行)。注意, Horse 类没有覆盖 GetAge(), 而只是继承它。

在第 46 行, Bird 类也被声明为从 Animal 派生而来, 其构造函数也接受年龄作为参数并使用它去初始化基类 Animal (第 62 行)。它也只是继承 GetAge() 而没有覆盖它。

在第 68 行, Pegasus 从 Bird 和 Horse 派生而来, 因此其继承链中有两个 Animal 类。如果要对 Pegasus 对象调用 GetAge() 方法, 而 Pegasus 类没有覆盖这个方法, 则必须明确或完全限定该方法。第 77 行覆盖 GetAge() 方法使其调用基类的同名方法, 从而解决了这种问题。

调用基类的同名方法的原因有两个: 明确要调用哪个基类的方法 (如本例所示); 先做一些工作, 然后再让基类的函数做另一些工作。有时候, 可能想先做一些工作然后调用基类的同名方法, 或先调用基类的同名方法, 并在该方法返回后再做一些工作。

从第 82 行开始的 Pegasus 构造函数接受 5 个参数: 颜色、高度 (类型为 HANDS)、是否迁徙、有多少人认为它存在和年龄。

在第 88 行, 构造函数使用颜色、高度和年龄初始化 Pegasus 的 Horse 部分; 第 89 行使用颜色、是否迁徙和年龄初始化 Bird 部分; 最后, 第 90 行初始化 itsNumberBelieves。

第 88 行对 Horse 构造函数的调用将调用第 39 行的实现。Horse 构造函数用年龄参数初始化 Pegasus 的 Horse 部分中的 Animal 部分, 然后初始化 Horse 的两个成员变量: itsColor 和 itsHeight。

第 89 行对 Bird 构造函数的调用将调用第 61 行的实现。这里也使用年龄参数来初始化 Bird 的 Animal 部分。

注意, Pegasus 的颜色参数被用来初始化 Bird 和 Horse 的成员变量。另外, 年龄被用来初始化 Horse 和 Bird 部分中的 Animal 的 itsAge。

**警告**

别忘了，显式地指定祖先类时将带来这样的风险：如果以后在当前类和祖先类之间插入一个新类，将导致调用跳过新的祖先类，直接进入原来的祖先类，这可能导致意外后果。

### 12.2.5 虚继承

在程序清单 12.5 中，Pegasus 尽力想明确使用哪个 Animal 基类。大多数时候，决定使用哪个是随意的，毕竟 Horse 和 Bird 有相同的基类。

可以告诉 C++，不想使用共同基类的两个副本，而只想要一个共同基类的副本，如图 12.3 所示。

为此，可让 Animal 成为 Horse 和 Bird 的虚基类。根本不用修改 Animal，对于 Horse 和 Bird 类，只需在其声明中使用关键字 virtual 即可。然而，对 Pegasus 类却要做重大修改。

通常，类的构造函数只初始化自己的变量及其基类，但虚继承的基类例外，它们由最后的派生类进行初始化。因此，Animal 不是由 Horse 和 Bird 初始化，而是由 Pegasus 初始化。Horse 和 Bird 必须在其构造函数中初始化 Animal，但创建 Pegasus 对象时，这些初始化将被忽略。

程序清单 12.6 重写了程序清单 12.5 以利用虚继承。

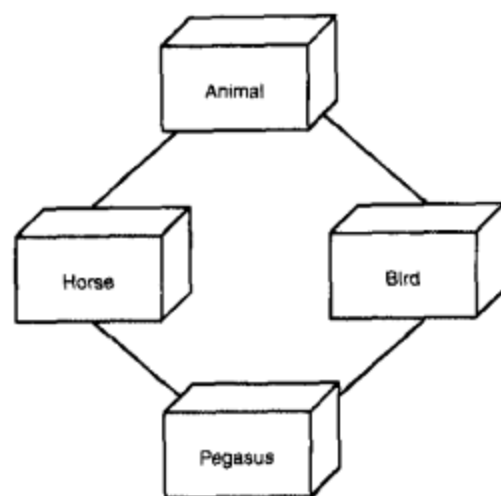


图 12.3 钻石形继承

#### 程序清单 12.6 使用虚继承

```

0: // Listing 12.6
1: // Virtual inheritance
2: #include <iostream>
3: using namespace std;
4:
5: typedef int HANDS;
6: enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };
7:
8: class Animal          // common base to both horse and bird
9: {
10: public:
11:     Animal(int);
12:     virtual ~Animal() { cout << "Animal destructor...\n"; }
13:     virtual int GetAge() const { return itsAge; }
14:     virtual void SetAge(int age) { itsAge = age; }
15: private:
16:     int itsAge;
17: };
18:
19: Animal::Animal(int age):
20:     itsAge(age)
21: {
22:     cout << "Animal constructor...\n";
23: }
24:
25: class Horse : virtual public Animal
26: {
27: public:
28:     Horse(COLOR color, HANDS height, int age);
29:     virtual ~Horse() { cout << "Horse destructor...\n"; }
30:     virtual void Whinny()const { cout << "Whinny!... "; }
31:     virtual HANDS GetHeight() const { return itsHeight; }
32:     virtual COLOR GetColor() const { return itsColor; }
33: protected:
34:     HANDS itsHeight;
35:     COLOR itsColor;
36: };
37:
38: Horse::Horse(COLOR color, HANDS height, int age):
39:     Animal(age),
40:     itsColor(color),itsHeight(height)
  
```

```

41: {
42:     cout << "Horse constructor...\n";
43: }
44:
45: class Bird : virtual public Animal
46: {
47:     public:
48:         Bird(COLOR color, bool migrates, int age);
49:         virtual ~Bird() {cout << "Bird destructor...\n"; }
50:         virtual void Chirp()const { cout << "Chirp... "; }
51:         virtual void Fly()const
52:             { cout << "I can fly! I can fly! I can fly! "; }
53:         virtual COLOR GetColor()const { return itsColor; }
54:         virtual bool GetMigration() const { return itsMigration; }
55:     protected:
56:         COLOR itsColor;
57:         bool itsMigration;
58: };
59:
60: Bird::Bird(COLOR color, bool migrates, int age):
61:     Animal(age),
62:     itsColor(color), itsMigration(migrates)
63: {
64:     cout << "Bird constructor...\n";
65: }
66:
67: class Pegasus : public Horse, public Bird
68: {
69:     public:
70:         void Chirp()const { Whinny(); }
71:         Pegasus(COLOR, HANDS, bool, long, int);
72:         virtual ~Pegasus() {cout << "Pegasus destructor...\n";}
73:         virtual long GetNumberBelievers() const
74:             { return itsNumberBelievers; }
75:         virtual COLOR GetColor()const { return Horse::itsColor; }
76:     private:
77:         long itsNumberBelievers;
78: };
79:
80: Pegasus::Pegasus(
81:     COLOR aColor,
82:     HANDS height,
83:     bool migrates,
84:     long NumBelieve,
85:     int age):
86:     Horse(aColor, height, age),
87:     Bird(aColor, migrates, age),
88:     Animal(age*2),
89:     itsNumberBelievers(NumBelieve)
90: {
91:     cout << "Pegasus constructor...\n";
92: }
93:
94: int main()
95: {
96:     Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
97:     int age = pPeg->GetAge();
98:     cout << "This pegasus is " << age << " years old.\n";
99:     delete pPeg;
100:    return 0;
101: }

```

#### ▼ 输出:

```

Animal constructor...
Horse constructor...
Bird constructor...
Pegasus constructor...
This pegasus is 4 years old.
Pegasus destructor...
Bird destructor...
Horse destructor...
Animal destructor...

```



▼ 分析:

第 25 行将 Horse 声明为从 Animal 虚继承而来;第 45 行对 Bird 做了同样声明。注意 Bird 和 Animal 的构造函数仍初始化其 Animal 部分。

Pegasus 从 Bird 和 Animal 派生而来,作为 Animal 最后的派生类,它也初始化 Animal。然而,调用的是 Pegasus 的初始化部分,而忽略 Bird 和 Horse 中对 Animal 构造函数的调用。之所以知道这一点,是因为传递的值为 2, Bird 和 Horse 将其传给 Animal,但 Pegasus 将乘以 2。第 98 行的打印输出表明,结果为 4。

Pegasus 不用明确调用 GetAge()引起的歧义,因为它只从 Animal 那里继承一个这样的函数。然而,Pegasus 仍必须明确对 GetColor()的调用,因为这个函数位于其两个基类而不是 Animal 类中。

声明虚继承类

为确保派生类只有一个共同基类的实例,可将中间类声明为从共同基类虚继承。

示例 1:

```
class Horse : virtual public Animal
class Bird : virtual public Animal
class Pegasus : public Horse, public Bird
```

示例 2:

```
class Schnauzer : virtual public Dog
class Poodle : virtual public Dog
class Schnoodle : public Schnauzer, public Poodle
```

## 12.2.6 多重继承存在的问题

虽然多重继承与单继承相比有很多优点,但很多 C++程序员却并不愿意使用它。他们提出的问题是,多重继承增加调试难度,开发多重继承类层次结构比开发单继承类层次结构更困难且风险更大,几乎所有用多重继承能完成的工作不用它也能完成。基于这些原因,诸如 Java 和 C#等语言不支持多重继承。

这些担心是有道理的,应避免在程序中引入不必要的复杂性。有些调试器对多重继承问题无能为力,有些设计在没必要的情况下使用多重继承,给其带来了不必要的复杂性。

应该	不应该
<p>当新类需要多个基类的函数和特征时应使用多重继承。</p> <p>当多个派生类只能有一个共同基类的实例时必须使用虚继承。</p> <p>使用虚基类时务必在最后的派生类中初始化共同基类。</p>	<p>在单继承可行的情况下不要使用多重继承。</p>

## 12.2.7 混合（功能）类

在多重继承和单继承之间的一种折衷方案是使用混合类 (mixin)。可以从 Animal 和 Displayable 类派生出 Horse 类, Displayable 只是添加一些将对象显示在屏幕上的方法。

混合（功能）类增加专用功能而不会增加大量方法或数据的类。

将功能类混合到派生类中的方法与其他类相同: 将派生类声明为以公有方式继承功能类。功能类

和其他类的唯一区别是：功能类没有或只有很少的数据。当然这种区分是不明确的，是一种表示只增加一些功能而不增加派生类复杂程度的方式。

对有些调试器来说，这使得处理混合继承对象比处理复杂的多重继承对象更容易。另外，在访问其他基类的数据时出现歧义的可能性更低。

例如，如果 Horse 从 Animal 和 Displayable 派生而来，而 Displayable 没有数据，这样，Horse 的全部数据都来自 Animal，而 Horse 中的函数来自两者。

#### 注意

混合继承的说法来自马萨诸塞州 Somerville 的一家冰激凌店，该店将糖果和蛋糕混合到基本的冰激凌口味中。这好像是某些在那里度暑假的面向对象程序员，尤其是使用面向对象编程语言 SCOOPS 的程序员们的比喻。

## 12.3 抽象数据类型

经常需要创建类层次结构。例如，可能创建 Shape 类，然后从其派生出 Rectangle 和 Circle。从 Rectangle 类，又可能派生出 Square 类，将其作为 Rectangle 的特例。

每个派生类都将覆盖 Draw()、GetArea() 方法等。程序清单 12.7 是 Shape 及其派生类 Circle 和 Rectangle 的简单实现。

程序清单 12.7 Shape 类

```
0: //Listing 12.7. Shape classes.
1:
2: #include <iostream>
3: using std::cout;
4: using std::cin;
5: using std::endl;
6:
7: class Shape
8: {
9:     public:
10:         Shape(){}
11:         virtual ~Shape(){}
12:         virtual double GetArea() { return -1; } // error value returned
13:         virtual double GetPerim() { return -1; }
14:         virtual void Draw() {}
15:     private:
16: };
17:
18: class Circle : public Shape
19: {
20:     public:
21:         Circle(int radius):itsRadius(radius){}
22:         ~Circle(){}
23:         double GetArea() { return 3 * itsRadius * itsRadius; }
24:         double GetPerim() { return 6 * itsRadius; }
25:         void Draw();
26:     private:
27:         int itsRadius;
28:         int itsCircumference;
29: };
30:
31: void Circle::Draw()
32: {
33:     cout << "Circle drawing routine here!\n";
34: }
35:
36:
37: class Rectangle : public Shape
38: {
39:     public:
40:         Rectangle(int len, int width):
41:             itsLength(len), itsWidth(width){}
```

```
42:     virtual ~Rectangle(){}
43:     virtual double GetArea() { return itsLength * itsWidth; }
44:     virtual double GetPerim() {return 2*itsLength + 2*itsWidth; }
45:     virtual int GetLength() { return itsLength; }
46:     virtual int GetWidth() { return itsWidth; }
47:     virtual void Draw();
48:     private:
49:         int itsWidth;
50:         int itsLength;
51: };
52:
53: void Rectangle::Draw()
54: {
55:     for (int i = 0; i<itsLength; i++)
56:     {
57:         for (int j = 0; j<itsWidth; j++)
58:             cout << "x ";
59:
60:         cout << "\n";
61:     }
62: }
63:
64: class Square : public Rectangle
65: {
66:     public:
67:         Square(int len);
68:         Square(int len, int width);
69:         ~Square(){}
70:         double GetPerim() {return 4 * GetLength();}
71: };
72:
73: Square::Square(int len):
74: Rectangle(len,len)
75: {}
76:
77: Square::Square(int len, int width):
78: Rectangle(len,width)
79: {
80:     if (GetLength() != GetWidth())
81:         cout << "Error, not a square... a Rectangle??\n";
82: }
83:
84: int main()
85: {
86:     int choice;
87:     bool fQuit = false;
88:     Shape * sp;
89:
90:     while ( !fQuit )
91:     {
92:         cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
93:         cin >> choice;
94:
95:         switch (choice)
96:         {
97:             case 0:    fQuit = true;
98:                         break;
99:             case 1: sp = new Circle(5);
100:                     break;
101:             case 2: sp = new Rectangle(4,6);
102:                     break;
103:             case 3: sp = new Square(5);
104:                     break;
105:             default:
106:                 cout <<"Please enter a number between 0 and 3"<<endl;
107:                 continue;
108:                 break;
109:         }
110:         if( !fQuit )
111:             sp->Draw();
112:         delete sp;
113:         sp = 0;
114:         cout << endl;
```

```

115:    }
116:    return 0;
117: }

```

### ▼ 输出:

```

(1)Circle (2)Rectangle (3)Square (0)Quit: 2
X X X X X X
X X X X X X
X X X X X X
X X X X X X

(1)Circle (2)Rectangle (3)Square (0)Quit:3
X X X X X
X X X X X
X X X X X
X X X X X
X X X X X

(1)Circle (2)Rectangle (3)Square (0)Quit:0

```

### ▼ 分析:

第 7~16 行声明了 Shape 类。GetArea() 和 GetPerim() 方法返回一个错误值，而 Draw() 不执行任何操作。绘制 Shape 究竟意味着什么呢？只有具体的形状（如圆、矩形等）才能绘制，Shape 作为一个抽象概念是不能绘制的。

在第 18~29 行，Circle 从 Shape 派生而来，并覆盖了 3 个虚方法。注意，没有理由使用关键字 virtual，因为这是其继承性的一部分；但这样做也没有害处，如 Rectangle 类声明中的第 43、44 和 47 行所示。作为提示（一种说明方式），加上关键字 virtual 是个不错的主意。

在第 64~71 行，Square 从 Rectangle 派生而来，它覆盖了 GetPerim() 方法，继承了 Rectangle 中定义的其他方法。

客户试图实例化 Shape 对象会很麻烦，也许应该使其不可能。毕竟，Shape 类只是为其派生类提供接口，因此它是一种抽象数据类型（ADT）。

在抽象类中，接口表示一种概念（如形状）而不是具体的对象（如圆）。在 C++ 中，抽象类只能用作其他类的基类，不能创建抽象类的实例。

## 12.3.1 纯虚函数

C++ 通过提供纯虚函数来支持创建抽象类。通过将虚函数初始化为 0 来将其声明为纯虚的，如下所示：

```
virtual void Draw() = 0;
```

在这个例子中，类有一个 Draw() 函数，但其实现为空，因此不能被调用。

任何包含一个或多个纯虚函数的类都是抽象类，因此不能对其实例化。事实上，试图对抽象类或从抽象类派生而来但没有实现其所有纯虚函数的类进行实例化都是非法的。试图这样做将导致编译错误。将纯虚函数放在类中向其客户指出了两点：

- 不要创建这个类的对象，而应从其派生；
- 务必覆盖从这个类继承的纯虚函数。

在从抽象类派生而来的类中，继承的纯虚函数仍是纯虚的，要实例化这种类的对象，必须覆盖每个纯虚函数。因此，如果 Rectangle 从 Shape 派生而来，而 Shape 有 3 个纯虚函数，Rectangle 必须覆盖这 3 个纯虚函数，否则它也将是抽象类。程序清单 12.8 重写了 Shape 类，使其成为一种抽象类。为节省篇幅，这里没有再次列出程序清单 12.7 中的其他代码。请用程序清单 12.8 中 Shape 的声明替换程序清单 12.7 中第 7~16 行的 Shape 声明，然后再次运行该程序。

## 程序清单 12.8 抽象类

```
0: //Listing 12.8 Abstract Classes
1:
2: class Shape
3: {
4:     public:
5:         Shape(){}
6:         ~Shape(){}
7:         virtual double GetArea() = 0;
8:         virtual double GetPerim()= 0;
9:         virtual void Draw() = 0;
10: };
```

## ▼ 输出:

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
• x x x x x
  x x x x x
  x x x x x
  x x x x x
  x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit: 0
```

## ▼ 分析:

正如读者看到的, 程序的运行情况完全相同。唯一的区别是, 现在不能创建 Shape 类对象了。

## 抽象数据类型

要将类声明为抽象类(也叫抽象数据类型), 在类声明中包含一个或多个纯虚函数即可。要将函数声明为纯虚的, 在声明函数后加上“=0”即可。

示例:

```
class Shape
{
    virtual void Draw() = 0;    // pure virtual
};
```

## 12.3.2 实现纯虚函数

通常, 不实现抽象基类的纯虚函数。由于不能创建抽象类的对象, 因此没有理由提供实现, 另外, 抽象类用作从其派生而来的类的接口定义。

然而, 可以给纯虚函数提供实现。这样, 就可以通过从抽象类派生而来的对象调用该函数, 该函数可能旨在给所有覆盖函数提供通用功能。程序清单 12.9 重写了程序清单 12.7, 这次将 Shape 声明为抽象类并提供了纯虚函数 Draw() 的实现。Circle 类覆盖了 Draw() 方法(必须这样做), 并调用了基类的 Draw() 方法以提供额外的功能。

在这个例子中, 额外的功能只是打印一条消息, 但可以想见, 基类可提供通用的绘图机制, 这也许是设置所有派生类都要用到的窗口。

## 程序清单 12.9 实现纯虚函数

```
0: //Listing 12.9 Implementing pure virtual functions
1:
2: #include <iostream>
3: using namespace std;
```



```

4:
5: class Shape
6: {
7:     public:
8:         Shape(){}
9:         virtual ~Shape(){}
10:        virtual double GetArea() = 0;
11:        virtual double GetPerim() = 0;
12:        virtual void Draw() = 0;
13:    private:
14: };
15:
16: void Shape::Draw()
17: {
18:     cout << "Abstract drawing mechanism!\n";
19: }
20:
21: class Circle : public Shape
22: {
23:     public:
24:         Circle(int radius):itsRadius(radius){}
25:         virtual ~Circle(){}
26:         double GetArea() { return 3.14 * itsRadius * itsRadius; }
27:         double GetPerim() { return 2 * 3.14 * itsRadius; }
28:         void Draw();
29:     private:
30:         int itsRadius;
31:         int itsCircumference;
32: };
33:
34: void Circle::Draw()
35: {
36:     cout << "Circle drawing routine here!\n";
37:     Shape::Draw();
38: }
39:
40:
41: class Rectangle : public Shape
42: {
43:     public:
44:         Rectangle(int len, int width):
45:             itsLength(len), itsWidth(width){}
46:         virtual ~Rectangle(){}
47:         double GetArea() { return itsLength * itsWidth; }
48:         double GetPerim() { return 2*itsLength + 2*itsWidth; }
49:         virtual int GetLength() { return itsLength; }
50:         virtual int GetWidth() { return itsWidth; }
51:         void Draw();
52:     private:
53:         int itsWidth;
54:         int itsLength;
55: };
56:
57: void Rectangle::Draw()
58: {
59:     for (int i = 0; i<itsLength; i++)
60:     {
61:         for (int j = 0; j<itsWidth; j++)
62:             cout << "x ";
63:
64:         cout << "\n";
65:     }
66:     Shape::Draw();
67: }
68:
69:
70: class Square : public Rectangle
71: {
72:     public:
73:         Square(int len);
74:         Square(int len, int width);
75:         virtual ~Square(){}
76:         double GetPerim() {return 4 * GetLength();}

```

```

77: };
78:
79: Square::Square(int len):
80:     Rectangle(len,len)
81: {}
82:
83: Square::Square(int len, int width):
84:     Rectangle(len,width)
85:
86: {
87:     if (GetLength() != GetWidth())
88:         cout << "Error, not a square... a Rectangle??\n";
89: }
90:
91: int main()
92: {
93:     int choice;
94:     bool fQuit = false;
95:     Shape * sp;
96:
97:     while (fQuit == false)
98:     {
99:         cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
100:        cin >> choice;
101:
102:        switch (choice)
103:        {
104:            case 1: sp = new Circle(5);
105:                    break;
106:            case 2: sp = new Rectangle(4,6);
107:                    break;
108:            case 3: sp = new Square (5);
109:                    break;
110:            default: fQuit = true;
111:                    break;
112:        }
113:        if (fQuit == false)
114:        {
115:            sp->Draw();
116:            delete sp;
117:            cout << endl;
118:        }
119:    }
120:    return 0;
121: }

```

#### ▼ 输出:

```

(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x
x x x x x
x x x x x
x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 0

```

#### ▼ 分析:

第 5~14 行声明了抽象类 Shape, 其 3 个存取器函数都被声明为纯虚函数。注意这不是必须的, 但是一种不错的做法。只要任何一个函数被声明为纯虚函数, 这个类就是抽象类。

GetArea() 和 GetPerim() 方法没有实现, 但第 16~19 行实现了 Draw() 方法。Circle 和 Rectangle 都覆盖了 Draw(), 它们都调用了同名的基类方法, 以利用基类提供的通用功能。

### 12.3.3 复杂的抽象层次结构

有时候，会从抽象类派生出其他抽象类。您可能想将一些继承而来的纯虚函数变成非纯虚函数，而保留其他的不变。

如果创建 `Animal` 类，可将 `Eat()`、`Sleep()`、`Move()` 和 `Reproduce()` 都声明为纯虚函数。您可能从 `Animal` 类派生出 `Mammal` 和 `Fish` 类。

经过考察后，您发现每种哺乳动物都以相同的方式进行繁殖，因此决定将 `Mammal::Reproduce()` 变成非纯虚函数，但保留 `Eat()`、`Sleep()` 和 `Move()` 为纯虚函数。

您从 `Mammal` 派生出 `Dog`，`Dog` 类必须覆盖和实现其他 3 个纯虚函数，以便能够创建 `Dog` 类对象。

作为类设计人员，您刚才的意思是，`Animals` 和 `Mammals` 不能实例化，但所有的哺乳动物类都可继承提供的 `Reproduce()` 方法而不覆盖它。程序清单 12.10 通过这些类的简化实现说明了这种技巧。

程序清单 12.10 从抽象类派生出其他抽象类

```

0: // Listing 12.10
1: // Deriving Abstract Classes from other Abstract Classes
2: #include <iostream>
3: using namespace std;
4:
5: enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };
6:
7: class Animal          // common base to both Mammal and Fish
8: {
9:     public:
10:         Animal(int);
11:         virtual ~Animal() { cout << "Animal destructor...\n"; }
12:         virtual int GetAge() const { return itsAge; }
13:         virtual void SetAge(int age) { itsAge = age; }
14:         virtual void Sleep() const = 0;
15:         virtual void Eat() const = 0;
16:         virtual void Reproduce() const = 0;
17:         virtual void Move() const = 0;
18:         virtual void Speak() const = 0;
19:     private:
20:         int itsAge;
21: };
22:
23: Animal::Animal(int age):
24:     itsAge(age)
25: {
26:     cout << "Animal constructor...\n";
27: }
28:
29: class Mammal : public Animal
30: {
31:     public:
32:         Mammal(int age):Animal(age)
33:         { cout << "Mammal constructor...\n"; }
34:         virtual ~Mammal() { cout << "Mammal destructor...\n"; }
35:         virtual void Reproduce() const
36:         { cout << "Mammal reproduction depicted...\n"; }
37: };
38:
39: class Fish : public Animal
40: {
41:     public:
42:         Fish(int age):Animal(age)
43:         { cout << "Fish constructor...\n"; }
44:         virtual ~Fish() { cout << "Fish destructor...\n"; }
45:         virtual void Sleep() const { cout << "fish snoring...\n"; }
46:         virtual void Eat() const { cout << "fish feeding...\n"; }
47:         virtual void Reproduce() const
48:         { cout << "fish laying eggs...\n"; }

```

```
49:     virtual void Move() const
50:     { cout << "fish swimming...\n"; }
51:     virtual void Speak() const { }
52: };
53:
54: class Horse : public Mammal
55: {
56:     public:
57:         Horse(int age, COLOR color ):
58:             Mammal(age), itsColor(color)
59:             { cout << "Horse constructor...\n"; }
60:         virtual ~Horse() { cout << "Horse destructor...\n"; }
61:         virtual void Speak()const { cout << "Whinny!... \n"; }
62:         virtual COLOR GetItsColor() const { return itsColor; }
63:         virtual void Sleep() const
64:             { cout << "Horse snoring...\n"; }
65:         virtual void Eat() const { cout << "Horse feeding...\n"; }
66:         virtual void Move() const { cout << "Horse running...\n"; }
67:
68:     protected:
69:         COLOR itsColor;
70: };
71:
72: class Dog : public Mammal
73: {
74:     public:
75:         Dog(int age, COLOR color ):
76:             Mammal(age), itsColor(color)
77:             { cout << "Dog constructor...\n"; }
78:         virtual ~Dog() { cout << "Dog destructor...\n"; }
79:         virtual void Speak()const { cout << "Whoof!... \n"; }
80:         virtual void Sleep() const { cout << "Dog snoring...\n"; }
81:         virtual void Eat() const { cout << "Dog eating...\n"; }
82:         virtual void Move() const { cout << "Dog running...\n"; }
83:         virtual void Reproduce() const
84:             { cout << "Dogs reproducing...\n"; }
85:
86:     protected:
87:         COLOR itsColor;
88: };
89:
90: int main()
91: {
92:     Animal *pAnimal=0;
93:     int choice;
94:     bool fQuit = false;
95:
96:     while (fQuit == false)
97:     {
98:         cout << "(1)Dog (2)Horse (3)Fish (0)Quit: ";
99:         cin >> choice;
100:
101:         switch (choice)
102:         {
103:             case 1: pAnimal = new Dog(5,Brown);
104:                     break;
105:             case 2: pAnimal = new Horse(4,Black);
106:                     break;
107:             case 3: pAnimal = new Fish (5);
108:                     break;
109:             default: fQuit = true;
110:                     break;
111:         }
112:         if (fQuit == false)
113:         {
114:             pAnimal->Speak();
115:             pAnimal->Eat();
116:             pAnimal->Reproduce();
117:             pAnimal->Move();
118:             pAnimal->Sleep();
119:             delete pAnimal;
120:             cout << endl;
```

```
121:     }
122:     }
123:     return 0;
124: }
```

▼ 输出:

```
(1)Dog (2)Horse (3)Bird (0)Quit: 1
Animal constructor...
Mammal constructor...
Dog constructor...
Whoof!...
Dog eating...
Dog reproducing...
Dog running...
Dog snoring...
Dog destructor...
Mammal destructor...
Animal destructor...

(1)Dog (2)Horse (3)Bird (0)Quit: 0
```

▼ 分析:

第 7~21 行声明了抽象类 `Animal`。`Animal` 有一个用于 `itsAge` 的非纯虚存取器函数，它们由所有动物类共享。还有 5 个纯虚函数：`Sleep()`、`Eat()`、`Reproduce()`、`Move()`和 `Speak()`。

在第 29~37 行，`Mammal` 被声明为从 `Animal` 派生而来，且没有新增任何数据，但覆盖了 `Reproduce()` 方法，为所有哺乳动物提供了通用的繁殖方式。`Fish` 必须覆盖 `Reproduce()`，因为它直接从 `Animal` 类派生而来，但不能采用哺乳动物的繁殖方式，这是在第 47~48 行实现的。

现在，哺乳动物类不必覆盖 `Reproduce()` 方法，但也可以选择这样做，就像 `Dog` 类声明中的第 83 行那样。`Fish`、`Horse` 和 `Dog` 都覆盖了其他的纯虚函数，这样便可以实例化相应类的对象了。

在该程序的 `main()` 函数中，将一个 `Animal` 指针依次指向各种派生类对象。调用虚函数时，将根据指针在运行阶段指向的对象，调用正确的派生类方法。试图实例化 `Animal` 或 `Mammal` 将导致编译错误，因为它们都是抽象类。

12.3.4 哪些类是抽象的

在一个程序中，`Animal` 类是抽象的，而在另一个程序可能不是。什么因素决定应将类声明为抽象的呢？

这个问题的答案不取决于现实世界的固有因素，而是由在程序中如何做合理决定的。假设您要编写一个描述农场或动物园的程序，可能将 `Animal` 声明为抽象类，但希望能实例化 `Dog` 类对象。

另一方面，如果要描述各种狗，可能将 `Dog` 声明为抽象类，且只实例化具体类型的狗：拾物狗、猎犬等。抽象层次取决于需要如何细分类型。

应该	不应该
务必使用抽象类给系列相关类通用的功能提供描述。	不要试图实例化抽象类。
务必将所有必须覆盖的函数声明为纯虚的。	

12.4 总结

本章介绍了如何克服单继承的某些局限性。读者知道了将函数沿继承层次结构向上提升以及沿继



承层次结构向下转换的风险。还学习了如何使用多重继承，多重继承会带来哪些问题，以及如何使用虚继承来解决这些问题。

本章还介绍了什么是抽象类以及如何使用纯虚函数来创建抽象类。学习了如何实现纯虚函数以及何时和为什么要这样做。

## 12.5 问与答

问：为什么判断对象的运行阶段类型是糟糕的？

答：因为这表明没有合理的构建继承层次结构，最好回过头去修复设计方案，而不是使用这种规避方式。

问：为什么向下转换是糟糕的？

答：如果以类型安全（type-safe）的方式进行，向下转换并没有什么不好。然而，向下转换可能破坏 C++ 的强类型检查，这是需要避免的。如果启用运行阶段类型识别，并对指针进行向下转换，这可能表明设计有问题。

问：为什么不将所有函数都声明为虚函数？

答：虚函数由虚函数表支持，后者会带来运行阶段开销，这种开销表现在程序的大小和性能方面。如果类非常小且预期不会有子类，则不应将其任何函数声明为虚函数。然而，当这种假设不成立时，应回过头去将祖先类的函数声明为虚函数，否则将导致意料之外的问题。

问：什么时候应将析构函数声明为虚函数？

答：在您认为类将被用作基类且基类指针将被用来访问派生类对象时。一种经验规则是，如果已经将类中的任何函数声明为虚函数，一定要将析构函数也声明为虚函数。

问：什么是接口？

答：接口是一种机制，指定了类应实现的方法以及这些方法的特征标。如果 IAnimal 是一个接口，而 CDog 是一个实现了接口 IAnimal 的类，则 CDog 必须实现接口 IAnimal 声明的方法。因此，接口相当于一种约定，而实现接口的类必须遵守该约定。换句话说，类要么实现整个接口，要么完全不实现。

问：如何使用 C++ 实现接口？

答：C++ 接口是只包含纯虚函数的抽象基类。继承抽象基类的类必须实现它声明的纯虚函数，并采用指定的特征标。

## 12.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 12.6.1 测验

1. 什么是向下转换？
2. 将功能向上提升是什么意思？

3. 如果圆角矩形有直边和圆角，且 RoundRect 类从 Rectangle 和 Circle 派生而来，而 Rectangle 和 Circle 类都是从 Shape 派生而来的，则创建一个 RoundRect 对象时将创建多少个 Shapes？
4. 如果 Horse 和 Bird 都采用公有虚继承从 Animal 派生而来，它们的构造函数会调用 Animal 的构造函数吗？如果 Pegasus 从 Horse 和 Bird 派生而来，它将如何调用 Animal 的构造函数？
5. 声明一个名为 Vehicle 的抽象类。
6. 如果基类为抽象类，它有 3 个纯虚函数，则其派生类需要覆盖其中的多少个函数？

### 12.6.2 练习

1. 声明 Jetplane 类，它从 Rocket 和 Airplane 派生而来。
2. 编写 Seven47 类的声明，它从练习 1 中的 Jetplane 类派生而来。
3. 编写类 Car 和 Bus 的声明，它们都从 Vehicle 类派生而来。将 Vehicle 类声明为包含有两个纯虚函数的抽象类。将 Car 和 Bus 声明为非抽象类。
4. 修改练习 3 编写的代码，将 Car 声明为抽象类并从 Car 派生出 SportsCar 和 Coupe。在 Car 类中，为 Vehicle 的一个纯虚函数提供实现，使其成为非纯虚函数。



## 第 13 章

# 运算符类型与运算符重载

虽然 C 语言编程中也有运算符，但 C++ 中的运算符有了新的含义，因为可对运算符进行编程使其能够对用户定义的类型（如类）进行操作，另外，还可重载运算符。

在本章中，您将学习：

- 使用关键字 `operator`
- 单目运算符与双目运算符
- 转换运算符
- 不能重新定义的运算符

### 13.1 C++ 中的运算符

从语法层面看，除使用关键字 `operator` 外，运算符与函数几乎没有差别。运算符声明看起来与函数声明极其相似：

```
return_type operator operator_symbol (...parameter list...);
```

其中 `operator_symbol` 是程序员可定义的几种运算符类型之一。可以是+（加）、&&（逻辑与）等。编译器可根据操作数区分运算符。那么，C++ 在支持函数的情况下为何还要提供运算符呢？

假设有一个字符串类 `CUsefulString`，并有该类的两个对象：

```
CUsefulString string1 ("Hello ");
```

```
CUsefulString string2 ("World!");
```

如果要将这两个字符拼接起来，下面两种方法哪种更方便、更直观呢？

方法 1：

```
CUsefulString stringSum;  
stringSum = string1 + string2;    // Hello World!
```

方法 2：

```
CUsefulString stringSum;  
stringSum = string1.Concatenate (string2); // Hello World!
```

显然，`CUsefulString` 实现的成员运算符+要优于成员函数 `Concatenate`。虽然两种方法的结果相同，但使用运算符的实现更直观、更易理解（因此更容易维护）。

因此，结论很明确：不要为简化类的实现而使用运算符；使用运算符旨在让类在他人看来更直观、更容易理解。

C++ 运算符分两大类：单目运算符与双目运算符。

### 13.2 单目运算符

顾名思义，单目运算符只对一个操作数进行操作。实现为全局函数或静态成员函数的单目运算符的典型定义如下：

```
return_type operator operator_type (parameter_type)
{
    // ... implementation
}
作为类成员的单目运算符的定义如下:
return_type operator operator_type ()
{
    // ... implementation
}
```

13.2.1 单目运算符的类型

可重载（或重新定义）的单目运算符如表 13.1 所示。

表 13.1 单目运算符

运算符	名称	运算符	名称
++	递增	&	取址
--	递减	~	求反
*	解除引用	+	正
->	成员选择	-	负
!	逻辑非	转换运算符	转换运算符

13.2.2 单目递增与单目递减运算符

下面通过对单目递增与单目递减运算符进行编程来解释单目运算符。假设有一个日历类 CDate，它包含 3 个整型成员，分别表示日、月和年。如果可以像对整型那样使用运算符++或--将 CDate 的值递增或递减就好了，为此需要在 CDate 类中实现单目递增和递减运算符，如程序清单 13.1 所示。

程序清单 13.1 一个处理日、月、年的日历类，可对日期执行递增操作

```
1: #include <iostream>
2:
3: class CDate
4: {
5: private:
6:     int m_nDay;      // Range: 1 - 30 (lets assume all months have 30 days!)
7:     int m_nMonth;    // Range: 1 - 12
8:     int m_nYear;
9:
10:    void AddDays (int nDaysToAdd)
11:    {
12:        m_nDay += nDaysToAdd;
13:
14:        if (m_nDay > 30)
15:        {
16:            AddMonths (m_nDay / 30);
17:
18:            m_nDay %= 30;    // rollover 30th -> 1st
19:        }
20:    }
21:
22:    void AddMonths (int nMonthsToAdd)
23:    {
24:        m_nMonth += nMonthsToAdd;
25:
26:        if (m_nMonth > 12)
27:        {
28:            AddYears (m_nMonth / 12);
29:
30:            m_nMonth %= 12;    // rollover dec -> jan
```

```
31:     }
32: }
33:
34: void AddYears (int m_nYearsToAdd)
35: {
36:     m_nYear += m_nYearsToAdd;
37: }
38:
39: public:
40:
41:     // Constructor that initializes the object to a day, month and year
42:     CDate (int nDay, int nMonth, int nYear)
43:         : m_nDay (nDay), m_nMonth (nMonth), m_nYear (nYear) {};
44:
45:     // Unary increment operator (prefix)
46:     CDate& operator ++ ()
47:     {
48:         AddDays (1);
49:         return *this;
50:     }
51:
52:     // postfix operator: differs from prefix in return-type and parameters
53:     CDate operator ++ (int)
54:     {
55:         // Create a copy of the current object, before incrementing day
56:         CDate mReturnDate (m_nDay, m_nMonth, m_nYear);
57:
58:         AddDays (1);
59:
60:         // Return the state before increment was performed
61:         return mReturnDate;
62:     }
63:
64:     void DisplayDate ()
65:     {
66:         std::cout << m_nDay << " / " << m_nMonth << " / " << m_nYear;
67:     }
68: };
69:
70: int main ()
71: {
72:     // Instantiate and initialize a date object to 25 May 2008
73:     CDate mDate (25, 6, 2008);
74:
75:     std::cout << "The date object is initialized to: ";
76:
77:     // Display initial date
78:     mDate.DisplayDate ();
79:     std::cout << std::endl;
80:
81:     // Applying the prefix increment operator
82:     ++ mDate;
83:
84:     std::cout << "Date after prefix-increment is: ";
85:
86:     // Display date after incrementing
87:     mDate.DisplayDate ();
88:     std::cout << std::endl;
89:
90:     return 0;
91: }
```

---

#### ▼ 输出:

```
The date object is initialized to: 25 / 5 / 2008
Date after prefix-increment is: 26 / 5 / 2008
```

---

#### ▼ 分析:

我们感兴趣的代码是第 45~50 行, 它们是单目前缀运算符 (++) 的实现, 该运算符让我们能够使用语句 ++mDate 将对象存储的日期后推一天, 如 main() 中的第 82 行所示。



从本质上说, `++mDate` 是由编译器通过调用 `operator ++` 实现的, 这相当于如下显式函数调用:

```
mDate.operator++ ();
```

注意, 第 52~62 行是后缀递增运算符的实现, 与前缀运算符实现的不同之处在于, 它接受一个输入参数, 并按值返回对象的副本。为方便进一步讨论, 程序清单 13.2 列出了后缀递增运算符的实现。

程序清单 13.2 后缀递增运算符

```
1: // postfix operator: differs from prefix in return-type and input param
2: CDate operator ++ (int)
3: {
4:     // Store a copy of the current state, before incrementing day
5:     CDate mReturnDate (m_nDay, m_nMonth, m_nYear);
6:
7:     IncrementDay ();
8:
9:     // Return the state before increment was performed
10:    return mReturnDate;
11: }
```

#### ▼ 分析:

从功能上说, 后缀递增运算符与前缀递增运算符的不同之处在于, `y = x++` 将导致 `y` 包含递增前的 `x` 值, 而 `y = ++x` 将导致 `y` 包含递增后的 `x` 值。因此, 后缀运算符 `++` 不按引用返回当前对象, 而按值返回递增前的对象副本。

因此, 也可像下面这样使用该后缀运算符:

```
// mDate initially contains 25 / 5 / 2008
```

```
// Use postfix increment
CDate mOldDate (mDate++);
```

```
// mDate now contains 26 / 5 / 2008
// mOldDate contains 25 / 5 / 2008
```

#### 注意

前缀与后缀递减运算符 (即运算符 `--`) 在语法上与递增运算符类似, 只是功能不同——将对象的值减 1。

### 13.2.3 解除引用运算符\*与成员选择运算符->的编程

这两个运算符可能在智能指针类编程中应用最广。智能指针是封装常规指针的类, 旨在通过管理所有权和复制问题简化内存管理。在有些情况下, 智能指针甚至能够提高应用程序的性能。智能指针将在第 26 章详细讨论。

下面来看一下程序清单 13.3 中 `std::auto_ptr` 的用法, 并理解它如何使用解除引用运算符\*使智能指针看起来像普通指针。

程序清单 13.3 `std::auto_ptr` 的用法

```
1: // Listing 13.3: How smart pointers that simulate normal pointer semantics
2: #include <memory>
3: #include <iostream>
4: using namespace std;
5:
6: class Dog
7: {
8: public:
9:     Dog () {};           // Constructor
10:    ~Dog () {};           // Destructor
11:
12:    void Bark ()
13:    {
14:        cout << "Bark! Bark!" << endl;
15:    }
16: };
```

```
17:
18: int main ()
19: {
20:     // a smart pointer equivalent of an int*
21:     auto_ptr<int> pSmartIntPtr (new int);
22:
23:     // Use this smart pointer like any normal pointer...
24:     *pSmartIntPtr = 25;
25:     cout << "pSmartIntPtr = " << *pSmartIntPtr << endl;
26:
27:     // a smart pointer equivalent of a Dog*
28:     auto_ptr<Dog> pSmartDog (new Dog);
29:
30:     // Use this smart pointer like any ordinary pointer
31:     pSmartDog->Bark ();
32:
33:     return 0;
34: }
```

### ▼ 分析:

注意到第 24~25 行使用了智能指针实例 `pSmartIntPtr`，第 31 行使用了 `pSmartDog`。如果读者没有注意到这两个智能指针的定义，很可能误认为 `pSmartIntPtr` 就是一个普通的 `int` 指针：

```
int* pSmartIntPtr = new int;
```

或认为 `pSmartDog` 是一个普通的 `Dog` 指针：

```
Dog* pSmartDog = new Dog;
```

然而，`pSmartIntPtr` 和 `pSmartDog` 都不是原始指针，而是标准类库的 `std` 命名空间提供的智能指针类 `auto_ptr` 的实例。`auto_ptr` 类的对象之所以可用作普通指针，是因为这个类实现了运算符 `*` 和 `->`。注意，这里有意没有使用 `delete` 释放整型，因为这项工作将由 `auto_ptr` 完成（因此称为智能指针）。

程序清单 13.4 是一个简单的智能指针类，它实现了解除引用运算符和成员选择运算符。

### 程序清单 13.4 一个简单的智能指针类

```
1: #template <typename T>
2: #class smart_pointer
3: {
4: private:
5:     T* m_pRawPointer;
6: public:
7:     // constructor
8:     smart_pointer (T* pData) : m_pRawPointer (pData) {}
9:
10:    // destructor
11:    ~smart_pointer () {delete m_pRawPointer ;}
12:
13:    // copy constructor
14:    smart_pointer (const smart_pointer & anotherSP);
15:
16:    // assignment operator
17:    smart_pointer& operator= (const smart_pointer& anotherSP);
18:
19:    // dereferencing operator
20:    T& operator* () const
21:    {
22:        return *(m_pRawPointer);
23:    }
24:
25:    // member selection operator
26:    T* operator-> () const
27:    {
28:        return m_pRawPointer;
29:    }
30: };
```

### ▼ 分析:

第 20~23 行实现了解除引用运算符，第 26~29 行实现了成员选择运算符，这使得可以像程序清

单 13.3 那样使用智能指针。注意到构造函数将提供的指针存储到一个局部变量中供前面提到的两个运算符使用。智能指针类的用户不会感到使用的是类而不是原始指针。

### 注意

与普通指针相比，除能够在指针离开作用域后释放其占用的内存外，智能指针还有很多其他功能，这将在第 26 章详细讨论。

如果读者对程序清单 13.3 中 `auto_ptr` 的用法感到好奇，可参考编译器或 IDE 提供的头文件 `<memory>` 中的 `auto_ptr` 实现，以便理解它在后台如何使其对象看起来像普通原始指针。

## 13.2.4 转换运算符的编程

如果要想让 `int nSomeNumber = Date(25, 5, 2008);` 成为一条有意义的语句，该如何办呢？也就是说，如果要想让每个 `Date` 对象都可转换为整数，以便将 `Date` 对象作为数字传递给只接受整型参数的其他模块，该如何办呢？

在 C++ 中，可通过自定义单目运算符来实现这种功能。典型的语法如下：

```
operator conversion_type();
```

因此，如果要编写一个转换运算符将日期转换为整数，可这样定义它：

```
operator int()
{
    // implementation
    return intValue;
}
```

这样不是很方便吗？来看看这是如何实现的，如程序清单 13.5 所示。

程序清单 13.5 使用转换运算符将 `CDate` 转换为整数

```
1: #include <iostream>
2:
3: class CDate
4: {
5: private:
6:     int m_nDay; // Range: 1 - 30 (lets assume all months have 30 days!)
7:     int m_nMonth; // Range: 1 - 12
8:     int m_nYear;
9:
10: public:
11:
12:     // Constructor that initializes the object to a day, month and year
13:     CDate (int nDay, int nMonth, int nYear)
14:         : m_nDay (nDay), m_nMonth (nMonth), m_nYear (nYear) {};
15:
16:     // Convert date object into an integer.
17:     operator int()
18:     {
19:         return ((m_nYear * 10000) + (m_nMonth * 100) + m_nDay);
20:     }
21:
22:     void DisplayDate ()
23:     {
24:         std::cout << m_nDay << " / " << m_nMonth << " / " << m_nYear;
25:     }
26: };
27:
28: int main ()
29: {
30:     // Instantiate and initialize a date object to 25 May 2008
31:     CDate mDate (25, 6, 2008);
32:
33:     std::cout << "The date object is initialized to: ";
34:
35:     // Display initial date
36:     mDate.DisplayDate ();
```

```
37:     std::cout << std::endl;
38:
39:     // Get the integer equivalent of the date
40:     int nDate = mDate;
41:
42:     std::cout << "The integer equivalent of the date is: " << nDate;
43:
44:     return 0;
45: }
```

▼ 输出:

The date object is initialized to: 25 / 6 / 2008  
The integer equivalent of the date is: 20080625

▼ 分析:

第 17~20 行定义的转换运算符 `int()` 只是返回日期对象对应的整数。现在,使用这种功能可将 `CDate` 对象转换为其他所需的任何类型。

就这里而言,另一个适用的转换运算符是将 `CDate` 对象转换为 `string` 类型,以使用字符串形式显示对象存储的日期:

```
operator std::string()
{
    // return a std::string containing the string equivalent of the date
}
```

### 13.3 双目运算符

对两个操作数进行操作的运算符称为双目运算符。以全局函数或静态成员函数的方式实现的双目运算符的定义如下:

`return_type operator_type (parameter1, parameter2);`

以类成员的方式实现的双目运算符的定义如下:

`return_type operator_type (parameter);`

以类成员的方式实现的双目运算符只接受一个参数,其原因是第二个参数通常是从类的属性获得的。

#### 13.3.1 双目运算符的类型

表 13.2 列出了可在 C++ 应用程序中重载或重新定义的双目运算符。

表 13.2 可重载的双目运算符

运算符	名称	运算符	名称
,	逗号	<	小于
!=	不等于	<<	左移
%	求模	<<=	左移并赋值
%=	求模并赋值	<=	小于或等于
&	按位与	=	赋值
&&	逻辑与	==	等于
&=	按位与并赋值	>	大于
*	乘	>=	大于或等于
*=	乘并赋值	>>	右移
+	加	>>=	右移并赋值
+=	加并赋值	^	异或
-	减	^=	异或并赋值
-=	减并赋值		按位或
->*	指向成员的指针	=	按位或并赋值
/	除		逻辑或
/=	除并赋值	[]	下标运算符

### 13.3.2 双目加与双目减运算符的编程

与递增/递减运算符类似，如果类实现了双目加和双目减运算符，便可将其对象加上或减去指定类型的值。再来看看日历类 CDate，虽然前面实现了将 CDate 递增以便前移一天的功能，但它还不支持增加 5 天的功能。为实现这种功能，需要实现双目加运算符，如程序清单 13.6 中的代码所示。

程序清单 13.6 实现了双目加运算符的日历类

```

1: #include <iostream>
2:
3: class CDate
4: {
5: private:
6:     int m_nDay;    // Range: 1 - 30 (lets assume all months have 30 days!)
7:     int m_nMonth;  // Range: 1 - 12
8:     int m_nYear;
9:
10:    void AddDays (int nDaysToAdd)
11:    {
12:        m_nDay += nDaysToAdd;
13:
14:        if (m_nDay > 30)
15:        {
16:            AddMonths (m_nDay / 30);
17:
18:            m_nDay %= 30;    // rollover 30th -> 1st
19:        }
20:    }
21:
22:    void AddMonths (int nMonthsToAdd)
23:    {
24:        m_nMonth += nMonthsToAdd;
25:
26:        if (m_nMonth > 12)
27:        {
28:            AddYears (m_nMonth / 12);
29:
30:            m_nMonth %= 12;    // rollover dec -> jan
31:        }
32:    }
33:
34:    void AddYears (int m_nYearsToAdd)
35:    {
36:        m_nYear += m_nYearsToAdd;
37:    }
38:
39: public:
40:
41:    // Constructor that initializes the object to a day, month and year
42:    CDate (int nDay, int nMonth, int nYear)
43:        : m_nDay (nDay), m_nMonth (nMonth), m_nYear (nYear) {};
44:
45:    CDate operator + (int nDaysToAdd)
46:    {
47:        CDate newDate (m_nDay, m_nMonth, m_nYear);
48:        newDate.AddDays (nDaysToAdd);
49:
50:        return newDate;
51:    }
52:
53:    void DisplayDate ()
54:    {
55:        std::cout << m_nDay << " / " << m_nMonth << " / " << m_nYear;
56:    }
57: };
58:
59: int main ()
60: {

```



```

61:    // Instantiate and initialize a date object to 25 May 2008
62:    CDate mDate (25, 6, 2008);
63:
64:    std::cout << "The date object is initialized to: ";
65:
66:    // Display initial date
67:    mDate.DisplayDate ();
68:    std::cout << std::endl;
69:
70:    std::cout << "Date after adding 10 days is: ";
71:
72:    // Adding 10 (days)...
73:    CDate datePlus10 (mDate + 10);
74:
75:    datePlus10.DisplayDate ();
76:
77:    return 0;
78: }

```

#### ▼ 输出:

```

The date object is initialized to: 25 / 6 / 2008
Date after adding 10 days is: 5 / 7 / 2008

```

#### ▼ 分析:

第 45~51 行是双目运算符+的实现,这使得能够采用第 73 行所示的语法。这行代码将一个 CDate 对象与整数 10 相加,它与下面的代码等价:

```
CDate datePlus10 (mDate.operator+ (10));
```

注意,这个例子没有实现双目减运算符(运算符-)。双目减运算符的调用语法与双目加运算符完全相同,但实现不同,因为要执行的是减法而不是加法。

### 13.3.3 运算符+=与-=的编程

加并赋值运算符支持语法 `a += b;`, 这让程序员可将对象 a 增加 b。这样,程序员可重载加并赋值运算符使其接受不同类型的参数 b。在程序清单 13.7 所示的示例中, b 是一个整型,但它也可以是 CDay 对象,用于指定将日历向前翻多少天。

#### 警告

警告: 下面的示例是不完整的,它只包含相关代码,即与加并赋值运算符相关的代码。其他代码在程序清单 13.6 中。

程序清单 13.7 使用加并赋值运算符将日历向前翻指定天数, 天数由整型输入参数指定

```

1: #class CDate
2: {
3: private:
4:     int m_nDay; // Range: 1 - 30 (lets assume all months have 30 days!)
5:     int m_nMonth; // Range: 1 - 12
6:     int m_nYear;
7:
8:     void AddDays (int nDaysToAdd);
9:     void AddMonths (int nMonthsToAdd);
10:    void AddYears (int m_nYearsToAdd);
11:
12: public:
13:
14:    // Constructor that initializes the object to a day, month and year
15:    CDate (int nDay, int nMonth, int nYear)
16:        : m_nDay (nDay), m_nMonth (nMonth), m_nYear (nYear) {};
17:
18:    // The addition-assignment operator    void operator += (int nDaysToAdd)
19:    {
20:        AddDays (nDaysToAdd);
21:    }
22:
23:    void DisplayDate ();
24: };

```

## ▼ 分析:

有了这个运算符后, 就可通过如下语法将日期增加指定的天数:

```
// Instantiate and initialize a date object to 25 May 2008
CDate mDate (25, 6, 2008);

// Adding 10 (days)...
mDate += 10;
```

除让加并赋值运算符接受整型输入参数外, 还可将该运算符重载使其接受其他类型的输入参数。

例如, 将 CDays 实例作为参数的加并赋值运算符如下:

```
// The addition-assignment operator that add a CDays to an existing date
void operator += (const CDays& mDaysToAdd)
{
    AddDays (mDaysToAdd.GetDays ());
}
```

## 注意

乘并赋值运算符 (\*=)、除并赋值运算符 (/=)、求模并赋值运算符 (%=)、减并赋值运算符 (-=)、左移并赋值运算符 (<<=)、右移并赋值运算符 (>>=)、异或并赋值运算符 (^=)、按位或并赋值运算符 |= 以及按位与并赋值运算符 (&=) 的语法都与程序清单 13.7 所示的加并赋值运算符类似。

虽然重载运算符的最终目标是让类更直观、更易于使用, 但很多时候实现这些运算符并没有意义。例如, 前面的日历类 CDate 绝对不会用到按位与并赋值运算符 &=。这个类的用户应该不会想通过 mDate &= 20; 等操作获得有用的结果。

## 13.3.4 重载比较运算符

如果像下面这样将前面的日期类的两个对象进行比较, 结果将如何呢?

```
if (mDate1 == mDate2)
{
    // Do something
}
else
{
    // Do something else
}
```

由于还没有定义等于运算符, 编译器将对这两个对象进行二进制比较, 并仅当它们完全相同时才返回 true。在有些情况下 (包括现在的 CDate 类), 这是可行的。然而, 如果 CDate 类由一个非静态字符串成员, 它包含字符串值 (char \*), 则比较结果可能不像期望的那样。在这种情况下, 对成员属性进行二进制比较时, 实际上将比较字符串指针, 而字符串指针并不相等 (即使指向的内容相同), 因此总是返回 false。

因此, 正确的做法是定义比较运算符。不等运算符的结果与等于运算符相反 (逻辑非)。程序清单 13.8 列出了日历类 CDate 定义的比较运算符。

程序清单 13.8 重载等于运算符和不等运算符

```
1: #include <iostream>
2: using namespace std;
3: class CDate
4: {
5: private:
6:     int m_nDay;    // Range: 1 - 30 (lets assume all months have 30 days!)
7:     int m_nMonth;  // Range: 1 - 12
8:     int m_nYear;
9:
10:    void AddDays (int nDaysToAdd);
11:    void AddMonths (int nMonthsToAdd);
12:    void AddYears (int m_nYearsToAdd);
13:
```

```
14: public:
15:
16:     // Constructor that initializes the object to a day, month and year
17:     CDate (int nDay, int nMonth, int nYear)
18:         : m_nDay (nDay), m_nMonth (nMonth), m_nYear (nYear) {};
19:
20:     void DisplayDate ()
21:     {
22:         cout << m_nDay << " / " << m_nMonth << " / " << m_nYear;
23:     }
24:
25:     // integer conversion operator
26:     operator int();
27:
28:     // equality operator that helps with: if (mDate1 == mDate2)...
29:     bool operator == (const CDate& mDateObj);
30:
31:     // overloaded equality operator that helps with: if (mDate == nInteger)
32:     bool operator == (int nDateNumber);
33:
34:     // inequality operator
35:     bool operator != (const CDate& mDateObj);
36:
37:     // overloaded inequality operator for integer types
38:     bool operator != (int nDateNumber);
39: };
40:
41: CDate::operator int()
42: {
43:     return ((m_nYear * 10000) + (m_nMonth * 100) + m_nDay);
44: }
45:
46: // equality operator that helps with if (mDate1 == mDate2)...
47: bool CDate::operator == (const CDate& mDateObj)
48: {
49:     return ( (mDateObj.m_nYear == m_nYear)
50:             && (mDateObj.m_nMonth == m_nMonth)
51:             && (mDateObj.m_nDay == m_nDay) );
52: }
53:
54: bool CDate::operator == (int nDateNumber)
55: {
56:     return nDateNumber == (int)*this;
57: }
58:
59: // inequality operator
60: bool CDate::operator != (const CDate& mDateObj)
61: {
62:     return !(this->operator==(mDateObj));
63: }
64:
65: bool CDate::operator != (int nDateNumber)
66: {
67:     return !(this->operator==(nDateNumber));
68: }
69:
70: void CDate::AddDays (int nDaysToAdd)
71: {
72:     m_nDay += nDaysToAdd;
73:
74:     if (m_nDay > 30)
75:     {
76:         AddMonths (m_nDay / 30);
77:
78:         m_nDay %= 30;    // rollover 30th -> 1st
79:     }
80: }
81: void CDate::AddMonths (int nMonthsToAdd)
82: {
83:     m_nMonth += nMonthsToAdd;
84:
85:     if (m_nMonth > 12)
86:     {
```

```

87:         AddYears (m_nMonth / 12);
88:
89:         m_nMonth %= 12;    // rollover dec -> jan
90:     }
91: }
92: void CDate::AddYears (int m_nYearsToAdd)
93: {
94:     m_nYear += m_nYearsToAdd;
95: }
96:
97: int main ()
98: {
99:     // Instantiate and initialize a date object to 25 May 2008
100:    CDate mDate1 (25, 6, 2008);
101:
102:    cout << "mDate1 contains: ";
103:
104:    // Display initial date
105:    mDate1.DisplayDate ();
106:    cout << endl;
107:
108:    CDate mDate2 (23, 5, 2009);
109:    cout << "mDate2 contains: ";
110:    mDate2.DisplayDate ();
111:    cout << endl;
112:
113:    // Use the inequality operator
114:    if (mDate2 != mDate1)
115:        cout << "The two dates are not equal... As expected!" << endl;
116:
117:    CDate mDate3 (23, 5, 2009);
118:    cout << "mDate3 contains: ";
119:    mDate3.DisplayDate ();
120:    cout << endl;
121:
122:    // Use the inequality operator
123:    if (mDate3 == mDate2)
124:        cout << "mDate3 and mDate2 are evaluated as equals" << endl;
125:
126:    // Get the integer equivalent of mDate3 using operator int()
127:    int nIntegerDate3 = mDate3;
128:
129:    cout << "The integer equivalent of mDate3 is:" << nIntegerDate3 << endl;
130:
131:    // Use overloaded operator== (for int comparison)
132:    if (mDate3 == nIntegerDate3)
133:        cout << "The integer and mDate3 are equivalent" << endl;
134:
135:    // Use overloaded operator != that accepts integers
136:    if (mDate1 != nIntegerDate3)
137:        cout << "The mDate1 is inequal to mDate3";
138:
139:    return 0;
140: }

```

#### ▼ 输出:

```

mDate1 contains: 25 / 6 / 2008
mDate2 contains: 23 / 5 / 2009
The two dates are not equal... As expected!
mDate3 contains: 23 / 5 / 2009
mDate3 and mDate2 are evaluated as equals using the equality operator
The integer equivalent of mDate3 is: 20090523
The integer and mDate3 are equivalent
The mDate1 is inequal to mDate3

```

#### ▼ 分析:

第 47~57 行实现两个等于运算符，其中一个用于 CDate 类型，另一个用于 int 类型。该运算符接受一个输入操作数，并将其与对象中的数据进行比较。进行什么样的比较由用户根据其应用程序的需求决定。第 60~68 行实现了不等运算符，它们只是将等于运算符的结果执行逻辑非操作。

### 13.3.5 重载运算符<、>、<=和>=

程序清单 13.8 所示的代码让 CDate 类足够聪明，能够判断两个 CDate 对象是否相等，还能判断一个整数是否与一个 CDate 对象包含的日期对应整数相等。

然而，如果要使用该类执行类似下面的条件检查，该如何办呢？

```
if (mDate1 < mDate2) { // do something }
```

或

```
if (mDate1 <= mDate2) { // do something }
```

或

```
if (mDate1 > mDate2) { // do something }
```

或

```
if (mDate >= mDate2) { // do something }
```

如果能够使用这个日历类来比较两个日期对象，以确定哪个日期在前，哪个日期在后，将很有用。编写这类的程序员应实现这种比较，让这个类对用户来说尽可能友好和直观，如程序清单 13.9 所示。

程序清单 13.9 实现运算符<、>、<=和>=

```
1: #include <iostream>
2:
3: class CDate
4: {
5: private:
6:     int m_nDay;    // Range: 1 - 30 (lets assume all months have 30 days!)
7:     int m_nMonth;  // Range: 1 - 12
8:     int m_nYear;
9:
10:    void AddDays (int nDaysToAdd);
11:    void AddMonths (int nMonthsToAdd);
12:    void AddYears (int m_nYearsToAdd);
13:
14: public:
15:
16:    // Constructor that initializes the object to a day, month and year
17:    CDate (int nDay, int nMonth, int nYear)
18:        : m_nDay (nDay), m_nMonth (nMonth), m_nYear (nYear) {}
19:
20:    void DisplayDate ()
21:    {
22:        cout << m_nDay << " / " << m_nMonth << " / " << m_nYear;
23:    }
24:
25:    bool operator < (const CDate& mDateObj) const;
26:    bool operator <= (const CDate& mDateObj) const;
27:
28:    bool operator > (const CDate& mDateObj) const;
29:    bool operator >= (const CDate& mDateObj) const;
30: };
31:
32: CDate::operator int() const
33: {
34:     return ((m_nYear * 10000) + (m_nMonth * 100) + m_nDay);
35: }
36:
37: bool CDate::operator < (const CDate& mDateObj) const
38: {
39:     return (this->operator int () < mDateObj.operator int ());
40: }
41:
42: bool CDate::operator > (const CDate& mDateObj) const
43: {
44:     return (this->operator int () > mDateObj.operator int ());
45: }
46:
47: bool CDate::operator <= (const CDate& mDateObj) const
```



```

48: {
49:     return (this->operator int () <= mDateObj.operator int ());
50: }
51:
52: bool CDate::operator >= (const CDate& mDateObj) const
53: {
54:     return (this->operator int () >= mDateObj.operator int ());
55: }
56:
57: // Pick the definition of other functions from listing 13.8
58:
59: int main ()
60: {
61:     // Instantiate and initialize a date object to 25 May 2008
62:     CDate mDate1 (25, 6, 2008);
63:     CDate mDate2 (23, 5, 2009);
64:     CDate mDate3 (23, 5, 2009);
65:
66:     cout << "mDate1 contains: ";
67:     mDate1.DisplayDate ();
68:     cout << endl;
69:
70:     cout << "mDate2 contains: ";
71:     mDate2.DisplayDate ();
72:     cout << endl;
73:
74:     cout << "mDate3 contains: ";
75:     mDate3.DisplayDate ();
76:     cout << endl;
77:
78:     // Use the operator <
79:     cout << "mDate3 < mDate2 is: ";
80:     cout << ((mDate3 < mDate2) ? "true" : "false") << endl;
81:
82:     // Use the operator <=
83:     cout << "mDate3 <= mDate2 is: ";
84:     cout << ((mDate3 <= mDate2) ? "true" : "false") << endl;
85:
86:     // Use operator >=
87:     cout << "mDate3 >= mDate1 is: ";
88:     cout << ((mDate3 >= mDate1) ? "true" : "false") << endl;
89:
90:     // Use operator >
91:     cout << "mDate1 > mDate3 is: ";
92:     cout << ((mDate1 > mDate3) ? "true" : "false") << endl;
93:
94:     return 0;
95: }

```

#### ▼ 输出:

```

mDate1 contains: 25 / 6 / 2008
mDate2 contains: 23 / 5 / 2009
mDate3 contains: 23 / 5 / 2009
mDate3 < mDate2 is: false
mDate3 <= mDate2 is: true
mDate3 >= mDate1 is: true
mDate1 > mDate3 is: false

```

#### ▼ 分析:

这里要讨论的运算符是在第 37~55 行实现的。注意到这些运算符是这样实现的：不是按年、月、日的顺序比较日期的每个元素，而通过使用转换运算符 `int()` 将工作简化为比较两个整数。

在 `main()` 函数的第 78~92 行使用了这些运算符，以演示这些运算符使得使用 `CDate` 类简单而直观。

#### 注意

在表 13.2 中，还有其他可重新定义或重载的双目运算符，本章不讨论它们，但其实现与这里讨论的运算符类似。

如果要在类中实现其他运算符（如逻辑运算符和按位运算符）以增强类的功能，则需要对其进行

编程。显然，像 CDate 这样的日历类不需要实现逻辑运算符，而执行字符串和数字操作的类总是需要实现它们。

在重载运算符或编写新运算符时，务必考虑类的目标和用途。

### 13.3.6 下标运算符

下标运算符使得能够像访问数组那样访问类，其典型语法如下：

```
return_type& operator [] (subscript_type& subscript);
```

编写存储整数的动态数组类 (CMyArray) 时，为确保能够像访问数组那样访问其内部元素，通常应这样实现下标运算符：

```
class CMyArray
{
    // ... other class members
public:
    int& operator [] (int nIndex)

    {
        // return the integer at position nIndex
    }
};
```

程序清单 13.10 是一个基本动态数组类的实现，并演示了下标运算符 ([]) 让用户能够使用普通数组语法来遍历该动态数组类中的元素。

程序清单 13.10 在动态数组类中实现下标运算符

```
1: #include <iostream>
2:
3: class CMyArray
4: {
5: private:
6:     int* m_pnInternalArray;
7:     int m_nNumElements;
8: public:
9:     CMyArray (int nNumElements);
10:    ~CMyArray ();
11:
12:    // declare a subscript operator
13:    int& operator [] (int nIndex);
14: };
15:
16: // subscript operator: allows direct access to an element given an index
17: int& CMyArray::operator [] (int nIndex)
18: {
19:     return m_pnInternalArray [nIndex];
20: }
21:
22: CMyArray::CMyArray (int nNumElements)
23: {
24:     m_pnInternalArray = new int [nNumElements];
25:     m_nNumElements = nNumElements;
26: }
27: CMyArray::~CMyArray ()
28: {
29:     delete [] m_pnInternalArray;
30: }
31:
32: int main ()
33: {
34:     // instantiate a dynamic array with 5 elements
35:     CMyArray mArray (5);
36:
37:     // write into the array using the subscript operator []
38:     mArray [0] = 25;
39:     mArray [1] = 20;
40:     mArray [2] = 15;
```

```

41:     mArray [3] = 10;
42:     mArray [4] = 5;
43:
44:     cout << "The contents of the array are: " << std::endl << "{";
45:
46:     // read from the dynamic array using the same subscript operator
47:     for (int nIndex = 0; nIndex < 5; ++ nIndex)
48:         std::cout << mArray [nIndex] << " ";
49:
50:     std::cout << "}";
51:
52:     return 0;
53: }

```

#### ▼ 输出:

```

The contents of the array are:
{25 20 15 10 5 }

```

#### ▼ 分析:

这个动态数组类非常简单，它并没有什么实际意义，只是为了演示下标运算符让您能够使用像数组式语法来读写对象的内容。下标运算符是在第 17~20 行定义的，它实现了 main() 函数的第 38~48 行使用的功能。

#### 注意

实现下标运算符时，可在程序清单 13.10 所示版本的基础上进行改进。这个版本只实现了一个下标运算符，它可用于读写动态数组中的元素。

然而，也可实现两个下标运算符，其中一个为 const 函数，另一个为非 const 函数：

```

int& operator [] (int nIndex);
int& operator [] (int nIndex) const;

```

编译器很聪明，能够在读取 CMyArray 对象时调用 const 函数，而在对 CMyArray 执行写入操作时调用非 const 函数。因此，如果愿意，可在两个下标函数中实现不同的功能。例如，一个运算符记录对容器的写入操作，而另一个记录对容器的读取操作。

## 13.4 operator()函数

operator()函数让对象像函数，它们应用于 STL（标准模板库）中，通常用于 STL 算法中。它们的用途包括决策，根据使用的操作数数量，这样的函数对象通常称为单目或双目谓词。下面分析一个非简单的函数对象，如程序清单 13.11 所示，以便理解使用如此有意思的名称的原因！

程序清单 13.11 一个使用 operator()实现的函数对象

```

1: #include <string>
2: #include <iostream>
3:
4: class CDisplay
5: {
6: public:
7:     void operator () (std::string strIn) const
8:     {
9:         std::cout << strIn << std::endl;
10:    }
11: };
12:
13: int main ()
14: {
15:     CDisplay mDisplayFuncObject;
16:
17:     // equivalent to mDisplayFuncObject.operator () ("Display this string!");
18:     mDisplayFuncObject ("Display this string!");
19:
20:     return 0;
21: }

```

▼ 输出:

Display this string!

▼ 分析:

第 7~10 行实现了 `operator()`，然后在 `main()` 函数的第 18 行使用了它。注意，之所以能够在第 18 行将对象 `mDisplayFuncObject` 用作函数，是因为编译器隐式地将它转换为对函数 `operator()` 的调用。因此，这个运算符也称为 `operator()` 函数，对象 `CDisplay` 也称为函数对象或 `functor`。

### 13.5 不能重新定义的运算符

虽然 C++ 提供了很大的灵活性，让程序员能够自定义运算符的行为让类更易于使用，但 C++ 也有所保留，它不允许程序员改变有些运算符的行为。表 13.3 列出了不能重新定义的运算符。

表 13.3 不能重载或重新定义的运算符

运算符	名称	运算符	名称
.	成员选择	?:	条件三目运算符
.*	指针成员选择	sizeof	获取对象/类类型的大小
::	作用域解析		

应该	不应该
<p>在定义不需要改变对象属性的成员运算符时，务必使用关键字 <code>const</code>。</p> <p>请牢记，在类中实现运算符可能增加程序员的工作量，但其他人使用它时将更容易。</p> <p>务必实现下标运算符的 <code>const</code> 版本和非 <code>const</code> 版本，以支持双向访问——一个用于读取，一个用于写入。</p>	<p>不要实现不必要的运算符。实现没人使用的运算符纯粹是浪费时间。</p>

### 13.6 总结

本章介绍了各种运算符以及如何重载它们让类更易于使用，还介绍了如何实现可帮助转换的运算符以及让对象可用作函数的运算符，最后介绍了 `.`、`.*`、`::`、`?:` 和 `sizeof` 等不能重新定义的运算符。

### 13.7 问与答

问：什么情况下应将运算符声明为 `const` 函数？

答：如果运算符不修改类的成员变量，最好将其声明为 `const` 函数。这不仅有助于编译器优化应用程序，还让编译器能够在有人试图在 `const` 运算符中修改类的成员时指出错误。

问：编写新类时，是否必须重载所有可重载的运算符？

答：不需要。定义运算符旨在让类更易于使用、更直观，但这并不意味着必须重载每个可重载的运算符。重载运算符时必须考虑应用程序。对于有些类（如用于数学方面的类），重载逻辑运算符和算术运算符可能会有所帮助，但对于处理字符串的类，重载用于转换类型的运算符可能会有所帮助。最后，还有一些类只能通过接口访问（出于设计的需要），重载运算符对这些类来说毫无意义。

问：编写智能指针类时需要定义哪些运算符？

答：智能指针类必须定义运算符\*和->。

## 13.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 13.8.1 测验

1. 编写两个版本的下标运算符，其中一个为 const，另一个为非 const 吗？这样做有什么好处？
2. 编写字符串类时必须定义哪些运算符？
3. 编写按位操作日期的类时，是否需要定义按位运算符？

### 13.8.2 练习

1. 为类 CDate 编写单目递减运算符声明。
2. 编写一个整型动态数组类（可以 CMyArray 为基础），它满足如下需求：
  - 如果两个数组的长度和内容都相同，则条件(mArray1 == mArray2)成立；
  - 如果 mArray1 的元素比 mArray2 少，或两者的长度相同但 mArray1 所有元素的总和小于 mArray2 所有元素的总和，则条件(mArray1 < mArray2)成立。





## 第 14 章

# 类型转换运算符

可改变对象解释方式的运算符称为类型转换运算符。

在本章中，您将学习：

- 为何需要类型转换运算符
- 为什么有些 C++ 程序员不喜欢传统的 C 风格类型转换
- 4 个 C++ 类型转换运算符
- 为什么 C++ 类型转换运算符并非总是最佳选择

### 14.1 什么是类型转换

类型转换是一种机制，让程序员能够暂时或永久性改变编译器对对象的解释。注意，这并不意味着程序员改变了对象本身，而只是改变了对对象的解释。

### 14.2 为何需要类型转换

如果 C++ 应用程序都编写得很完善，其处于类型是安全的且是强类型的世界，则没有必要进行类型转换，也不需要类型转换运算符。然而，在现实世界中，不同模块往往由不同的人编写，而使用不同开发环境的厂商需要协作。因此，程序员经常需要让编译器按其所需的方式解释数据，让应用程序能够成功编译并正确执行。

来看一个真实的例子：虽然 C++ 编译器支持 `bool`，但有很多年前使用 C 语言编写的库仍在被使用。这些针对 C 语言编译器编写的库必须依赖于整型来保存布尔值，因此对这些编译器来说，`bool` 类型类似于下面这样：

```
typedef unsigned short BOOL;
```

而返回布尔值的函数可能这样声明：

```
BOOL IsX ();
```

如果要在新的应用程序中使用一个这样的库，而该应用程序将使用最新的 C++ 编译器进行编译，则程序员必须让其使用的 C++ 编译器能够理解数据类型 `bool`，同时让库能够理解数据类型 `bool`。为此，可使用类型转换：

```
bool bCPPResult = (bool)IsX ();    // C-Style cast
```

在 C++ 的发展过程中，不断有新的 C++ 类型转换运算符出现，这导致 C++ 编程社区分裂成两个阵营：一个阵营继续在其 C++ 应用程序中使用 C 风格类型转换，另一个阵营转而使用 C++ 编译器引入的类型转换关键字。前一个阵营认为，C++ 类型转换难以使用，且有时候功能变化不大，只有理论意义。后一个阵营则显然由 C++ 语法纯粹论者组成，他们通过指出 C 风格类型转换的缺陷以支持其论点。

在现实世界中，这两个观点各行其道，读者最好通过阅读本章了解每种风格的优缺点，然后形成自己的见解。

## 14.3 为何有些 C++ 程序员不喜欢 C 风格类型转换

C++ 程序员在赞颂这门编程语言时提到的优点之一是类型安全。实际上，大多数 C++ 编译器都不会让下面这样的语句通过编译：

```
char* pszString = "Hello World!";  
int* pBuf = pszString;    // error: cannot convert char* to unsigned char*
```

这是非常正确的！

当前，C++ 编译器仍需向后兼容以确保遗留代码能够通过编译，因此支持下面这样的语法：

```
int* pBuf = (int*)pszString;    // Cast one problem away to create a bigger one!
```

然而，C 风格类型转换实际上强迫编译器根据程序员的选择来解释目标对象。就上述代码而言，程序员并不认为编译器报告错误是合理的，因此强迫编译器遵从自己的意愿。然而，对不希望类型转换破坏其倡导的类型安全的 C++ 程序员来说，这是无法接受的。

## 14.4 C++ 类型转换运算符

虽然类型转换有缺点，但也不能抛弃类型转换的概念。在很多情况下，类型转换是合理的需求，可解决重要的兼容性问题。C++ 提供了一种新的类型转换运算符，专门用于基于继承的情形，这种情形在 C 语言编程中并不存在。

4 个 C++ 类型转换运算符如下：

- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `const_cast`

这 4 个类型转换运算符的使用语法相同：

```
destination_type result = cast_type <destination_type> (object_to_be_casted);
```

### 14.4.1 使用 `static_cast`

`static_cast` 用于在相关类型的指针之间进行转换，还可显式地执行标准数据类型的类型转换——这种转换原本将自动或隐式地进行。用于指针时，`static_cast` 实现了基本的编译阶段检查，确保指针被转换为相关类型。这改进了 C 风格类型转换，在 C 语言中，可将指向一个对象的指针转换为完全不相关的类型，而编译器不会报错。使用 `static_cast` 可将指针向上转换为基类类型，也可向下转换为派生类型，如下面的示例代码所示：

```
CBase* pBase = new CDerived ();    // construct a CDerived object  
CDerived* pDerived = static_cast<CDerived*>(pBase);    // ok!  
  
// CUnrelated is not related to CBase via any inheritance hierarchy  
CUnrelated* pUnrelated = static_cast<CUnrelated*>(pBase); // Error
```

//The cast above is not permitted as types are unrelated

然而，`static_cast` 只验证指针类型是否相关，而不会执行任何运行阶段检查。因此，程序员可使用 `static_cast` 编写如下代码，而编译器不会报错：

```
CBase* pBase = new CBase ();  
CDerived* pDerived = static_cast<CDerived*>(pBase); // Still no errors!
```

其中 `pDerived` 实际上指向一个不完整的 `CDerived` 对象，因为它指向的对象实际上是 `CBase()` 类型。由于 `static_cast` 只在编译阶段检查转换类型是否相关，而不执行运行阶段检查，因此 `pDerived->SomeDerivedClassFunction()` 能够通过编译，但在运行阶段可能导致意外结果。

除用于向上和向下转换外, `static_cast` 还可在很多情况下将隐式类型转换为显式类型, 以引起程序员或代码阅读者的注意:

```
double dPi = 3.14159265;  
int nNum = static_cast<int>(dPi); // Making an otherwise implicit cast, explicit
```

在上述代码中, 使用 `nNum = dPi` 将获得同样的效果, 但使用 `static_cast` 可让代码阅读者注意到这里使用了类型转换, 并指出 (对知道 `static_cast` 的人而言) 编译器根据编译阶段可用的信息进行了必要的调整, 以便执行所需的类型转换。

## 14.4.2 使用 `dynamic_cast` 和运行阶段类型识别

顾名思义, 与静态类型转换相反, 动态类型转换在运行阶段 (即应用程序运行时) 执行类型转换。可检查 `dynamic_cast` 操作的结果以判断类型转换是否成功。使用 `dynamic_cast` 运算符的典型语法如下:

```
destination_type* pDest = dynamic_cast <class_type*> (pSource);  
  
if (pDest) // Check for success of the casting operation  
    pDest->CallFunc ();
```

例如:

```
CBase* pBase = new CDerived();  
  
// Perform a downcast  
CDerived* pDerived = dynamic_cast <CDerived*> (pBase);  
  
if (pDerived) // Check for success of the cast  
    pDerived->CallDerivedClassFunction ();
```

如上述代码所示, 给定一个指向基类的对象, 程序员可使用 `dynamic_cast` 进行类型转换, 并在使用指针前检查指针指向的目标对象的类型。在上述示例代码中, 目标对象的类型显然是 `CDerived`, 因此这些代码只有演示价值。然而, 情况并非总是如此。给定一个基类对象, 程序员可能不确定它属于哪种派生类类型, 这时可使用 `dynamic_cast` 在运行阶段判断其类型, 并在安全时使用转换后的指针。因此, 这种在运行阶段识别对象类型的机制称为运行阶段类型识别 (runtime type identification, RTTI), 如程序清单 14.1 所示。

程序清单 14.1 有助于判断 `Animal` 对象是 `Cat` 还是 `Dog` 的动态转换

```
1: #include <iostream>  
2: using namespace std;  
3: class CAnimal  
4: {  
5: public:  
6:     virtual void Speak () = 0;  
7: };  
8:  
9: class CDog : public CAnimal  
10: {  
11: public:  
12:     void WagTail () {cout << "Dog: I wagged my tail!" << endl;}  
13:  
14:     void Speak () {cout << "Dog: Bow-Wow!" << endl;}  
15: };  
16:  
17: class CCat : public CAnimal  
18: {  
19: public:  
20:     void CatchMice () {cout << "Cat: I caught a mouse!" << endl;}  
21:  
22:     void Speak () {cout << "Cat: Meow!" << endl;}  
23: };  
24:  
25: void DetermineType (CAnimal* pAnimal);  
26:  
27: int main ()
```

```
28: {
29:     // pAnimal1 points to a Dog object
30:     CAnimal* pAnimal1 = new CDog ();
31:
32:     // pAnimal2 points to a Cat object
33:     CAnimal* pAnimal2 = new CCat ();
34:
35:     cout << "Using dynamic_cast to determine type of Animal 1" << endl;
36:     DetermineType (pAnimal1);
37:
38:     cout << "Using dynamic_cast to determine type of Animal 2" << endl;
39:     DetermineType (pAnimal2);
40:
41:     // Use the virtual function overridden by the subclasses to prove type
42:     cout << "Verifying type: Asking Animal 1 to speak!" << endl;
43:     pAnimal1->Speak ();
44:
45:     cout << "Verifying type: Asking pAnimal 2 to speak!" << endl;
46:     pAnimal2->Speak ();
47:
48:     return 0;
49: }
50:
51: void DetermineType (CAnimal* pAnimal)
52: {
53:     CDog* pDog = dynamic_cast <CDog*>(pAnimal);
54:     if (pDog)
55:     {
56:         cout << "The animal is a dog!" << endl;
57:
58:         // Call the derived class' function
59:         pDog->WagTail ();
60:     }
61:
62:     CCat* pCat = dynamic_cast <CCat*>(pAnimal);
63:     if (pCat)
64:     {
65:         cout << "The animal is a cat!" << endl;
66:
67:         pCat->CatchMice ();
68:     }
69: }
```

#### ▼ 输出:

```
Using dynamic_cast to determine type of Animal 1
The animal is a dog!
Dog: I wagged my tail!
Using dynamic_cast to determine type of Animal 2
The animal is a cat!
Cat: I caught a mouse!
Verifying type: Asking Animal 1 to speak!
Dog: Bow-Wow!
Verifying type: Asking pAnimal 2 to speak!
Cat: Meow!
```

#### ▼ 分析:

派生类 CDog 和 CCat 继承了抽象基类 CAnimal, 为此, 这两个类必须分别将纯虚函数 Speak() 定义为狗吠和猫叫。函数 DetermineType 使用 dynamic\_cast 转换 CAnimal 指针以区分狗和猫, 这是在第 51~69 行实现的。确定动物类型后, 该函数使用这个指针调用相应派生类的函数, 即使用指向 CDog 对象的指针调用 WagTail(), 并使用指向 CCat 对象的指针调用 CatchMice()。在 main() 函数中, 使用 DetermineType 执行运行阶段类型识别, 并通过 CAnimal 指针调用虚函数 Speak()。

### 14.4.3 使用 reinterpret\_cast

reinterpret\_cast 是 C++ 中与 C 风格类型转换最接近的类型转换运算符。它让程序员能够将一种对象

类型转换为另一种，不管它们是否相关；也就是说，它使用如下所示的语法强制重新解释类型：

```
CBase * pBase = new CBase ();
CUnrelated * pUnrelated = reinterpret_cast<CUnrelated*>(pBase);
```

// The code above is not good programming, even if it compiles!

这种类型转换实际上是强制编译器接受 `static_cast` 通常不允许的类型转换，通常用于低级程序（如驱动程序），在这种程序中，需要将数据转换为 API 能够接受的简单类型（例如，有些 API 只能使用字节流，即 `unsigned char*`）：

```
CSomeClass* pObject = new CSomeClass ();
// Need to send the object as a byte-stream...
unsigned char* pBytes = reinterpret_cast<unsigned char*>(pObject);
```

上述代码使用的类型转换并没有改变源对象的二进制表示，但让编译器允许程序员访问 `CSomeClass` 对象包含的各个字节。由于其他 C++ 类型转换运算符都不允许执行这样的转换，因此使用 `reinterpret_cast` 时用户将收到类型转换不安全（不可移植）的警告。

#### 提示

应尽量避免使用 `reinterpret_cast`。

### 14.4.4 使用 `const_cast`

`const_cast` 让程序员能够关闭对象的访问修饰符 `const`。您可能会问：为何要进行这种转换？在理想情况下，程序员将经常在正确的地方使用关键字 `const`。不幸的是，现实世界并不理想，经常可以看到该使用 `const` 的地方没有使用，如下面的代码所示：

```
CSomeClass
{
public:
    // ...
    void DisplayMembers ();    // ought to be a const member
};
```

在下面的函数中，以 `const` 引用的方式传递 `mData` 对象显然是正确的。毕竟，显示函数应该是只读的，不应调用非 `const` 成员函数，即不应调用能够修改对象状态的函数。然而，`DisplayMembers()` 本应为 `const` 的，但却没有这样定义。如果 `CSomeClass` 归您所有，且源代码受您控制，则可对 `DisplayMembers()` 进行修改。然而，在很多情况下，它可能属于第三方库，无法对其进行修改。在这种情况下，`const_cast` 将是您的救星。

```
void DisplayAllData (const CSomeClass& mData)
{
    mData.DisplayMembers ();    // Compile failure
    // reason for failure: call to a non-const member using a const reference
}
```

在这种情况下，调用 `DisplayMembers()` 的语法如下：

```
void DisplayAllData (const CSomeClass& mData)
{
    CSomeClass& refData = const_cast<CSomeClass&>(mData);
    refData.DisplayMembers();    // Allowed!
}
```

#### 注意

应将使用 `const_cast` 来调用非 `const` 函数作为最后的手段。一般而言，使用 `const_cast` 来修改 `const` 对象可能导致不可预料的行为。

另外，`const_cast` 也可用于指针：

```
void DisplayAllData (const CSomeClass* pData)
{
    // pData->DisplayMembers(); Error: attempt to invoke a non-const function!
    CSomeClass* pCastedData = const_cast<CSomeClass*>(pData);
    pCastedData->DisplayMembers();    // Allowed!
}
```



## 14.5 C++类型转换运算符存在的问题

并非所有人都喜欢使用 C++ 类型转换，即使那些 C++ 拥趸也如此。其理由很多，从语法繁琐而不够直观到显得多余。

来比较一下下面的代码：

```
double dPi = 3.14159265;

// C++ style cast: static_cast
int nNum = static_cast<int>(dPi);    // result: nNum is 3

// C-style cast
int nNum2 = (int)dPi;                // result: nNum is 3

// leave casting to the compiler
int nNum3 = dPi;                    // result: nNum is 3. No errors!
```

在这 3 种方法中，程序员得到的结果都相同。在实际情况下，第 2 种方法可能最常见，其次是第 3 种，但几乎没有人使用第 1 种方法。无论采用哪种方法，编译器都足够聪明，能够正确地进行类型转换。这让人觉得类型转换运算符将降低代码的可读性。

同样，static\_cast 的其他用途也可使用 C 风格类型转换进行处理，且更简单：

```
// using static_cast
CDerived* pDerived = static_cast<CDerived*>(pBase);

// But, this works just as well...
CDerived* pDerivedSimple = (CDerived*)pBase;
```

因此，使用 static\_cast 的优点常常被其拙劣的语法所掩盖。Bjarne Stroustrup 准确地描述了这种境况：“由于 static\_cast 如此拙劣且难以输入，因此您在使用它之前很可能会三思。这很不错，因为类型转换在现代 C++ 中是最容易避免的。”

再来看其他运算符。在不能使用 static\_cast 时，可使用 reinterpret\_cast 强制进行转换；同样，可以使用 const\_cast 修改访问修饰符 const。因此，在现代 C++ 中，除 dynamic\_cast 外的类型转换都是可以避免的。仅当需要满足遗留应用程序的需求时，才需要使用其他类型转换运算符。在这种情况下，程序员通常倾向于使用 C 风格类型转换而不是 C++ 类型转换运算符。重要的是，应尽量避免使用类型转换；而一旦使用类型转换，务必要知道幕后发生的情况。

## 14.6 总结

本章介绍了各种 C++ 类型转换运算符以及支持和反对类型转换运算符的根据。一般而言，应避免使用类型转换。

## 14.7 问与答

问：是否可使用 const\_cast 对指向常量对象的指针或引用进行类型转换，以便修改常量对象的内容？

答：不要这样做。这样做的结果是不确定的，也绝不是您希望的。

问：我需要一个 CBird\*，但只有一个 CDog\*。编译器不允许将指向 CDog 对象的指针用作 CBird\*。然而，当我使用 reinterpret\_cast 将 CDog\* 转换为 CBird\* 时，编译器并不报错。看起来可使用这个指针来调用 CBird 的成员函数 Fly()，可以这样做吗？

答：绝对不要这样做。reinterpret\_cast 只改变对指针的解释，并不改变指向的对象（它还是 Dog）。对 CDog 对象调用 Fly() 函数将得不到所需的结果，还可能导致应用程序出现故障。

问：我有一个 CBase 指针 pBase，它指向一个 CDerived 对象。我确信 pBase 指向的是一个 CDerived 对象，是否还需要使用 dynamic\_cast？

答：由于您确定指向的是 CDerived 对象，因此可使用 static\_cast 提高运行性能。

问：C++ 提供了类型转换运算符，但却建议尽量不使用它们。这是为什么？

答：您家里备有阿司匹林，却不会把它当饭吃。仅当真正需要时才使用类型转换。

## 14.8 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学的知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 测验

1. 您有一个基类对象指针 pBase，要确定其类型是 CDerived1 还是 CDerived2，应使用哪种类型转换？
2. 假设您有一个指向对象的 const 引用，并试图通过它调用一个您编写的公有成员函数，但编译器不允许您这样做，因为该函数不是 const 成员。您将修改这个函数还是使用 const\_cast？
3. 判断对错：仅在不能使用 static\_cast 时才应使用 reinterpret\_cast，这种类型转换是必须和安全的。
4. 判断对错：优秀的编译器将自动执行很多基于 static\_cast 的类型转换，尤其是简单数据类型之间的转换。



## 第 15 章

# 宏和模板简介

现在，读者应对基本的 C++ 语法有深入认识，能够理解使用 C++ 编写的程序，为学习有助于高效地编写程序的语言特性做好了准备。

在本章中，您将学习：

- 预处理器简介
- 关键字 `#define` 与宏
- 模板简介
- 编写函数模板和类模板
- 宏和模板之间的区别

### 15.1 预处理器与编译器

每当您编译 C++ 文件时，编译器都将首先对其进行预处理。编译器中在实际编译代码前对其进行处理的单元称为预处理器。

在 C++ 中，程序员可使用预处理指令控制应用程序在编译阶段的行为，每条预处理指令都以 `#` 打头，这些指令影响源代码中的文本，指定在预编译阶段如何修改它们，然后编译器再对修改后的输出进行编译。

读者通过编译指令 `#include` 见过预处理器的影响，该指令将包含文件的内容加入源代码中。

### 15.2 预处理器指令 `#define`

可使用命令 `#define` 来定义字符串替换，下面的代码指示预处理器将所有的 “BIG” 替换为 512：

```
#define BIG 512
```

这不是 C++ 意义上的字符串。源代码中所有的 “BIG” 都将被替换为 “512”。对于如下源代码：

```
#define BIG 512
```

```
int myArray[BIG];
```

经预处理器处理后将变成：

```
int myArray[512];
```

注意到编译器已经将 BIG 替换为 512。

#### 提示

需要指出的是，预处理器进行基于纯粹的文本替换，按前面的定义将声明数组时使用的 BIG 替换为 512。它并不执行上下文相关检查，以核实用于替换 BIG 的值是否适合插入定义数组的代码中。

也可能将 BIG 定义为字符串 Tiny，这时预处理器提供给编译器的数组声明将为：

```
int myarray ["Tiny"]
```

这当然会导致编译错误。使用预处理器不是类型安全的，后面将看到这种问题带来的严重后果。

## 15.3 宏函数

编译指令 `#define` 也可以用于创建宏函数。宏函数是使用 `#define` 创建的符号，它像函数那样能够接受参数。预处理器将用指定的参数值替换宏函数中的替换字符串。例如，如果将宏 `TWICE` 定义为：

```
#define TWICE(x) ( (x) * 2 )
```

然后程序中包含如下代码：

```
TWICE(4)
```

则预处理器将把 `TWICE(4)` 替换为 `((4)*2)`，结果为 8。

宏可以接受多个参数，每个参数都可在替换文本中多次出现。两个常见的宏是 `MAX` 和 `MIN`：

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
```

```
#define MIN(x,y) ( (x) < (y) ? (x) : (y) )
```

注意，在宏的定义中，参数列表的左括号必须紧跟在宏名后面，中间不能有空格。预处理器不像编译器那样允许使用空白，如果有空格，将像本章前面介绍的那样使用标准替换。

例如，如果编写了如下宏：

```
#define MAX (x,y) ( (x) > (y) ? (x) : (y) )
```

然后像下面这样使用 `MAX`：

```
int x = 5, y = 7, z;
```

```
z = MAX(x,y);
```

中间代码将为：

```
int x = 5, y = 7, z;
```

```
z = (x,y) ( (x) > (y) ? (x) : (y) )(x,y)
```

这执行了简单的文本替换，而不是调用宏函数。因此，符号 `MAX` 被替换为 `(x,y) ( (x) > (y) ? (x) : (y) )`，然后是 `MAX` 后面的 `(x,y)`。

如果将 `MAX` 和 `(x,y)` 之间的空格删除，中间代码将为：

```
int x = 5, y = 7, z;
```

```
a = ( (5) > (7) ? (5) : (7) );
```

### 15.3.1 为什么要使用括号

您可能会问，为什么前面介绍的很多宏中包含那么多括号。预处理器并不要求在替换字符串中用括号将参数括起，但将复杂的值传递给宏时，括号有助于避免副作用。例如，如果将 `MAX` 定义为：

```
#define MAX(x,y) x > y ? x : y
```

然后调用该宏并将 5 和 7 传递给它，它将按期望的方式运行。然而，如果传递的是更复杂的表达式，将不能得到预期的结果，如程序清单 15.1 所示。

程序清单 15.1 在宏中使用括号

```
0: // Listing 15.1 Macro Expansion
1: #include <iostream>
2: using namespace std;
3:
4: #define CUBE(a) ( (a) * (a) * (a) )
5: #define THREE(a) a * a * a
6:
7: int main()
8: {
9:     long x = 5;
10:    long y = CUBE(x);
11:    long z = THREE(x);
12:
13:    cout << "y: " << y << endl;
14:    cout << "z: " << z << endl;
15:
16:    long a = 5, b = 7;
17:    y = CUBE(a+b);
```

```

18:     z = THREE(a+b);
19:
20:     cout << "y: " << y << endl;
21:     cout << "z: " << z << endl;
22:     return 0;
23: }

```

#### ▼ 输出:

```

y: 125
z: 125
y: 1728
z: 82

```

#### ▼ 分析:

第 4 行定义了宏 CUBE，其中在每次使用参数 a 时都将其用括号括起。第 5 行定义了宏 THREE，但没有使用括号。

在第 10 行和第 11 行首次调用这两个宏时，传递的参数值为 5，这两个宏都运行正常。CUBE(5) 展开为 ((5)\*(5)\*(5))，结果为 125；THREE(5) 展开为 5\*5\*5，结果也是 125。

在第 16~18 行第二次调用它们时，参数为 5+7。在这种情况下，CUBE(5+7) 展开为 ((5+7)\*(5+7)\*(5+7))，这相当于 ((12)\*(12)\*(12))，结果为 1728。而 THREE(5+7) 展开为 5+7\*5+7\*5+7。由于乘法的优先级高于加法，因此这相当于 5+(7\*5)+(7\*5)+7，即 5+(35)+(35)+7，结果为 82。正如读者看到的，由于没有使用括号导致了错误，5+7 的三倍实际上为 36！

## 15.3.2 宏与类型安全问题

回到前面那个简单的宏函数：

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
```

显然，这个宏用于比较相似的类型。

然而，由于宏只是预处理器提供的一种文本替换功能，因此宏处理后的文本（或变量）没有任何类型安全，即不进行类型检查。

在理想的情况下，希望对泛型函数进行限制使其只比较类似的对象。在本章后面，读者将看到如何使用模板提供这种功能。

## 15.3.3 宏与函数及模板之比较

在 C++ 中，宏存在 4 个问题。首先，如果宏很大将令人迷惑，因为所有宏都必须在一行中定义。可以使用反斜杠 (\) 扩展到下一行，但大型宏将很快变得难以管理。

其次，宏在每次被使用时都按内联方式展开。这意味着，如果一个宏被使用了 12 次，程序中将包含 12 次替换结果，而不像函数调用那样只出现一次。另一方面，它们的速度通常比函数调用快，因为没有函数调用的开销。

宏按内联方式扩展导致了第三个问题，即宏不能出现在编译器使用的中间源代码中，因此在大多数调试器中无法看到，这使得宏的调试非常棘手。

最后一个也是最严重的问题是，宏不是类型安全的。虽然使用宏时几乎可以提供任何类型的参数，这很方便，但这完全破坏了 C++ 的强类型功能，因此被 C++ 程序员视为瘟疫。

## 15.3.4 内联函数

通常可以不使用宏，而声明一个内联函数。例如，程序清单 15.2 创建了内联函数 Cube()，该函数



的功能与程序清单 15.1 中的 CUBE 宏相同，但以类型安全的方式完成其工作。

程序清单 15.2 使用内联函数而不是宏

```
0: #include <iostream>
1: using namespace std;
2:
3: inline unsigned long Square(unsigned long a) { return a * a; }
4: inline unsigned long Cube(unsigned long a)
5:     { return a * a * a; }
6: int main()
7: {
8:     unsigned long x=1 ;
9:     for (;;)
10:    {
11:        cout << "Enter a number (0 to quit): ";
12:        cin >> x;
13:        if (x == 0)
14:            break;
15:        cout << "You entered: " << x;
16:        cout << ". Square(" << x << "): ";
17:        cout << Square(x);
18:        cout<< ". Cube(" << x << "): ";
19:        cout << Cube(x) << "." << endl;
20:    }
21:    return 0;
22: }
```

▼ 输出：

```
Enter a number (0 to quit): 1
You entered: 1. Square(1): 1. Cube(1): 1.
Enter a number (0 to quit): 2
You entered: 2. Square(2): 4. Cube(2): 8.
Enter a number (0 to quit): 3
You entered: 3. Square(3): 9. Cube(3): 27.
Enter a number (0 to quit): 4
You entered: 4. Square(4): 16. Cube(4): 64.
Enter a number (0 to quit): 5
You entered: 5. Square(5): 25. Cube(5): 125.
Enter a number (0 to quit): 6
You entered: 6. Square(6): 36. Cube(6): 216.
Enter a number (0 to quit): 0
```

▼ 分析：

第 3 行和第 4 行定义了两个内联函数：Square()和 Cube()。它们都被声明为内联的，因此像宏一样，每次调用时都将就地展开，没有函数调用的开销。

注意，展开为内联函数意味着在调用函数的地方（如第 17 行）放置函数的代码。由于没有进行函数调用，因此没有将返回地址和参数压入堆栈的开销。

第 17 行调用函数 Square()，第 19 行调用函数 Cube()。由于它们是内联函数，因此就像第 16~19 行被编写成下面这样：

```
16:     cout << ". Square(" << x << "): " ;
17:     cout << x * x ;
18:     cout << ". Cube(" << x << "): " ;
19:     cout << x * x * x << "." << endl;
```

应该	不应该
宏名应大写。这是一种通用约定，如果不这样做，其他程序员将感到迷惑。 应用括号将宏中每个参数括起。	不要让宏产生副作用。不要在宏中递增变量或给变量赋值。 在使用 const 变量可行的情况下，不要使用 #define 来定义常量。

## 15.4 模板简介

模板可能是 C++ 语言中最强大却最少被使用的特性之一。

在 C++ 中，模板让程序员能够定义一种适用于不同类型对象的行为。这听起来有点像宏（参见前面用于判断两个数中哪个更大的简单宏 MAX），但宏不是类型安全的，而模板是类型安全的。

### 15.4.1 模板声明语法

模板声明以关键字 `template` 打头，接下来是一个类型参数列表。这种声明的格式如下：

```
template <parameter list>
...template declaration..
```

下面来分析一个模板声明，它与前面讨论的宏 MAX 等价：

```
template <typename objectType>
objectType & GetMax (const objectType & value1, const objectType & value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
};
```

关键字 `template` 标志着模板声明的开始，接下来是模板参数列表。该参数列表包含关键字 `typename`，它定义了模板参数 `objectType`，`objectType` 是一个占位符，针对对象实例化模板时，将使用对象的类型替换它。模板声明包含要声明的模式。

下面是一个使用该模板的示例：

```
int nInteger1 = 25;
int nInteger2 = 40;
int nMaxValue = GetMax <int> (nInteger1, nInteger2);
double dDouble1 = 1.1;
double dDouble2 = 1.001;
double dMaxValue = GetMax <double> (nDouble1, nDouble2);
```

注意，调用 `GetMax` 时使用了 `<int>`，这将模板参数 `objectType` 指定为 `int`。上述代码将导致编译器生成模板函数 `GetMax` 的两个版本，如下所示：

```
const int & GetMax (const int& value1, const int& value2)
{
    //...
}
const double & GetMax (const double& value1, const double& value2)
{
    // ...
}
```

然而，实际上调用模板函数时并非一定要指定类型，因此下面的函数调用没有任何问题：

```
int nMaxValue = GetMax (nInteger1, nInteger2);
```

在这种情况下，编译器很聪明，知道这是针对整型调用模板函数。然而，这只适用于模板函数，而不适用于模板类。

### 15.4.2 各种类型的模板声明

模板声明可以是：

- 函数的声明或定义（如前面所示）；
- 类的定义或声明；
- 类模板的成员函数或成员类的声明或定义；
- 类模板的静态数据成员的定义；

- 类或类模板的成员模板的定义。

### 15.4.3 模板类

模板类是模板化的 C++ 类，C++ 类在第 10 章介绍过。

下面是一个简单的模板类，它只有一个模板参数 T：

```
template <typename T>
class CMyFirstTemplateClass
{
public:
    void SetVariable (T& newValue) { m_Value = newValue; };

    T& GetValue () {return m_Value;};

private:
    T m_Value;
};
```

类 CMyFirstTemplate 用于保存一个类型为 T 的变量，该变量的类型是在使用模板时指定的。下面来看该模板类的一种用法：

```
CMyFirstTemplate <int> mHoldInteger; // Template instantiation
mHoldInteger.SetValue (5);
std::cout << "The value stored is: " << mHoldInteger.GetValue ();
```

这里使用该模板类来存储和检索类型为 int 的对象，即使用 int 类型的模板参数实例化 Template 类。

同样，这个类也可以用于处理字符串，其用法类似：

```
CMyFirstTemplate <char*> mHoldString;
mHoldInteger.SetValue ("Sample string");
std::cout << "The value stored is: " << mHoldInteger.GetValue ();
```

因此，这个类定义了一种模式，并可针对不同的数据类型实现这种模式。

### 15.4.4 模板的实例化和具体化

对于模板，术语实例化的含义稍有不同。用于类时，实例化通常指的是根据类创建对象。

但用于模板时，实例化指的是根据模板声明以及一个或多个参数创建特定的类型。

因此，对于下面的模板声明：

```
template <typename T>
class CTemplateClass
{
    T m_member;
};
```

使用该模板时将编写这样的代码：

```
CTemplateClass <int> mIntTemplate;
```

这种实例化创建的特定类型称为具体化。

### 15.4.5 模板与类型安全

与 MAX 宏功能相同的模板是类型安全的，如果这样调用 GetMax：

```
int nMaxValue = GetMax (nInteger, "Some string");
```

这种没有意义的调用将导致编译错误，而宏 MAX 将能够通过编译，甚至连警告也没有。

### 15.4.6 使用多个参数声明模板

模板参数列表包含多个参数，参数之间用逗号分隔。因此，如果要声明一个泛型类用于存储两个类型可能不同的对象，可以使用如下所示的代码（这个模板类包含两个模板参数）：

```

template <typename T1, typename T2>
class CHoldsPair
{
private:
    T1 m_Value1;
    T2 m_Value2;
public:
    // Constructor that initializes member variables
    CHoldsPair (const T1& value1, const T2& value2)
    {
        m_Value1 = value1;
        m_Value2 = value2;
    };
    // ... Other function declarations
};

```

在这里，类 CHoldsPair 接受两个模板参数，参数名分别为 T1 和 T2。可使用这个类来存储两个类型相同或不同的对象，如下所示：

```

// A template instantiation that pairs an int with a double
CHoldsPair <int, double> pairIntDouble (6, 1.99);

// A template instantiation that pairs an int with an int
CHoldsPair <int, int> pairIntDouble (6, 500);

```

### 15.4.7 使用默认参数来声明模板

可以修改前面的 CHoldsPair <...>，将模板参数的默认类型指定为 int：

```

template <typename T1=int, typename T2=int>
class CHoldsPair
{
    // ... Function declarations
};

```

这与给函数指定默认参数值极其类似，只是这里指定的是默认类型。

这样，前述第二种 CHoldsPair 用法可以简写为：

```

// A template instantiation that pairs an int with an int (default type)
CHoldsPair <> pairIntDouble (6, 500);

```

### 15.4.8 一个模板示例

下面使用前面讨论的 CHoldsPair 模板来进行开发，如程序清单 15.3。

程序清单 15.3 前面定义的模板类

---

```

1: // Template class with default template parameters
2: template <typename T1=int, typename T2=double>
3: class CHoldsPair
4: {
5: private:
6:     T1 m_Value1;
7:     T2 m_Value2;
8: public:
9:     // Constructor that initializes member variables
10:    CHoldsPair (const T1& value1, const T2& value2)
11:    {
12:        m_Value1 = value1;
13:        m_Value2 = value2;
14:    };
15:
16:    // Accessor functions
17:    const T1 & GetFirstValue ()
18:    {
19:        return m_Value1;
20:    };
21:
22:    const T2& GetSecondValue ()

```

资源解密

PDG

```
23: {  
24:     return m_Value2;  
25: };  
26: };
```

#### ▼ 分析:

可以看到,第2行有一个模板参数列表,它定义了两个参数,这两个参数的默认类型分别为 `int` 和 `double`。另外,还有存取器函数 `GetFirstValue()` 和 `GetSecondValue()`,它们用于查询对象的值。这个类看起来像一个框架,用户可根据要处理的数据类型对其进行具体化,如程序清单 15.4 所示。

#### 程序清单 15.4 使用模板类

```
1: // LISTING 15.4 - Sample Usage of the Template Class CHoldsPair  
2: int main ()  
3: {  
4:     using namespace std;  
5:  
6:     // Two instantiations of template CHoldsPair -  
7:     CHoldsPair <> mIntFloatPair (300, 10.09);  
8:     CHoldsPair<short,char*>mShortStringPair(25,"Learn templates, love C++");  
9:  
10:    // Output values contained in the first object...  
11:    cout << "The first object contains -" << endl;  
12:    cout << "Value 1: " << mIntFloatPair.GetFirstValue () << endl;  
13:    cout << "Value 2: " << mIntFloatPair.GetSecondValue () << endl;  
14:  
15:    // Output values contained in the second object...  
16:    cout << "The second object contains -" << endl;  
17:    cout << "Value 1: " << mShortStringPair.GetFirstValue () << endl;  
18:    cout << "Value 2: " << mShortStringPair.GetSecondValue ();  
19:  
20:    return 0;  
21: }
```

#### ▼ 输出:

```
The first object contains -  
Value 1: 300  
Value 2: 10.09  
The second object contains -  
Value 1: 25  
Value 2: Learn templates, love C++
```

#### ▼ 分析:

这个简单程序演示了如何声明模板类 `CHoldsPair` 来存储两个值,这两个值的类型取决于模板的参数列表。注意到存取器函数 `GetFirstValue()` 和 `GetSecondValue()` 将根据模板实例化语法返回正确的对象类型。

至此,在 `CHoldsPair` 中定义了一种模式,可通过重用该模式针对不同的变量类型实现相同的逻辑。因此,使用模板可提高代码的可复用性。

### 15.4.9 在实际 C++ 编程中使用模板

模板最重要也是最强大的应用是在标准模板库 (STL) 中。STL 由包含通用实用类和算法的模板类组成。这些 STL 模板类让您能够实现动态数组、链表以及包含关键字-值对的容器,而 `sort` 等算法可应用于这些容器,从而对容器包含的数据进行处理。

前面介绍的模板语法有助于读者使用本章后面将详细介绍的 STL 容器和函数;而更深入地理解 STL 将有助于使用 STL 中经过测试的可靠实现,从而编写出更高效的 C++ 程序,还有助于避免在模板细节上浪费时间。



应该	不应该
<p>当一个概念适用于不同类（或基本数据类型）的对象时，应使用模板来实现它。</p> <p>使用模板函数的参数来限制模板实例的类型，从而确保类型安全。</p> <p>应根据类型覆盖模板函数，以具体化模板的行为。</p>	<p>不要就此结束对模板的学习。本章只介绍了模板的部分用途，对模板的详细介绍超出了本书的范围。</p> <p>即使没有完全理解如何创建自己的模板也不要着急，知道如何使用模板更重要。</p>

## 15.5 总结

本章更详细地介绍了预处理器。每当您运行编译器时，预处理器都将首先运行，对`#define`等指令进行转换。

预处理器执行文本替换，但在使用宏时替换将比较复杂。通过使用宏函数，可根据在编译阶段传递给宏的参数进行复杂的文本替换。将宏中的每个参数放在括号内以确保进行正确的替换，这很重要。

模板有助于编写可重用的代码，它向开发人员提供了一种可用于不同数据类型的模式。模板可以取代宏，且是类型安全的。学习本章介绍的模板知识后，便为学习如何使用标准模板库（STL）做好了准备！

## 15.6 问与答

问：既然 C++ 提供了比预处理器更好的替代品，为什么还支持预处理器呢？

答：首先，C++ 向后与 C 语言兼容，因此必须支持 C 语言的所有重要特性。其次，预处理器的一些用法在 C++ 还是经常使用的，如多重包含防范（inclusion guard）。

问：在可以使用常规函数的情况下为什么要使用宏函数？

答：宏函数被展开为内联函数，用于替换需要重复输入而变化很小的命令。然而，模板通常是更好的选择。

问：如何判断应使用宏还是内联函数？

答：尽可能使用内联函数。虽然宏提供了字符替换、字符串化和拼接功能，但它们不是类型安全的，还可能导致代码难以维护。

问：在调试期间，除使用预处理器打印中间值外，还有什么方法？

答：最佳的方法是在调试器中使用监视（有时称为跟踪）语句。有关监视语句的更详细信息，请参阅编译器或调试器的文档。

问：在使用宏可行时为何要使用模板？

答：模板是类型安全的且是 C++ 语言内置的，编译器将对其进行检查——至少在您实例化模板类以创建变量时会这样做。

问：模板函数的参数化类型与普通函数的参数之间有何不同？

答：普通函数（非模板）接受它能够进行处理的参数，而模板函数让您能够指定函数参数的类型。也就是说，可向函数传递一个 Type 数组，然后根据变量（类或类型的实例）的定义确定 Type。

## 15.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄清这些答案。

### 15.7.1 测验

1. 什么是多重包含防范 (inclusion guard) ?
2. `#define debug 0` 与 `#undef debug` 之间的区别何在?
3. 如果使用参数 4 调用下面的宏，结果将是多少?  
`#define HALVE(x) x / 2`
4. 如果用 `10+10` 调用问题 3 中的 `HALVE` 宏，结果将是多少?
5. 如何修改 `HALVE` 宏以避免得到错误的结果?
6. 模板与宏之间有何不同?
7. 模板参数与 (非模板) 函数参数之间有何不同?
8. STL 表示什么? 为何 STL 至关重要?

### 15.7.2 练习

1. 编写一个将两个数相加的宏。
2. 编写一个模板实现练习 1 中宏的功能。
3. 实现模板函数 `swap`，它交换两个变量的值。
4. 查错：您将如何改进下面的宏使其计算输入值的 1/4?  
`#define QUARTER(x) (x / 4)`
5. 编写一个简单的模板类，它存储两个数组，数组的类型是通过模板参数列表指定的。数组包含 10 个元素，模板类应包含存取器函数，可用于操作数组元素。







## 第三部分

# 学习标准模板库（STL）

第 16 章 标准模板库简介

第 17 章 STL string 类

第 18 章 STL 动态数组类

第 19 章 STL list

第 20 章 STL set 与 multiset

第 21 章 STL map 和 multimap

## 第 16 章

# 标准模板库简介

简单地说，标准模板库（STL）是一组模板类和函数，向程序员提供了：

- 存储信息的容器；
- 访问容器存储的信息的迭代器；
- 操作容器内容的算法。

本章概述 STL 的这 3 个重要方面。

### 16.1 STL 容器

容器是用于存储数据的 STL 类，STL 提供了两种类型的容器类：

- 顺序容器；
- 关联容器。

#### 16.1.1 顺序容器

顾名思义，顺序容器按顺序存储数据，如数组和列表。顺序容器具有插入速度快但查找操作较慢的特征。

STL 顺序容器包括：

- `std::vector`，操作与动态数组一样，在最后插入数据；
- `std::deque`，与 `std::vector` 类似，但也允许在开头插入新元素；
- `std::list`，操作与链表一样。

STL `vector` 类与数组类似，它允许随机访问元素，即可使用下标运算符指定元素在向量中的位置（索引），从而直接访问或操作元素。另外，STL `vector` 是动态数组，因此能够根据应用程序在运行阶段的需求自动调整长度。为保留数组能够根据位置随机访问元素的特征，大多数 STL `vector` 实现都将所有元素存储在连续的存储单元中，因此需要调整长度的 `vector`，通常将降低应用程序的性能，这取决于它包含的对象类型。

可将 STL `list` 类视为普通链表的 STL 实现。虽然 `list` 中的元素不能够像 STL `vector` 中的元素那样随机访问，但 `list` 可使用不连续的内存块组织元素，因此它不像 `vector` 那样需要给内部数组重新分配内存，进而导致性能问题。

#### 16.1.2 关联容器

关联容器按指定的顺序存储数据，就像词典一样。这将降低插入数据的速度，而在查询方面有很大的优势。



STL 提供的关联容器包括：

- `std::set`，按排序排列的唯一值列表；
- `std::map`，存储键-值对，并根据唯一的键将键-值对排序；
- `std::multiset`，与 `set` 类似，但允许存储多个值相同的项，即值不需要是唯一的；
- `std::multimap`，与 `map` 类似，但不要求键是唯一的。

可通过谓词函数编程定制 STL 容器的排序标准。

提示

有些 STL 实现也支持关联容器 `hash_set`、`hash_multiset`、`hash_map` 和 `hash_multimap`。这些容器有更好的元素搜索性能，因为其元素访问时间为常量，不依赖于容器大小。相比之下，在遵循标准的关联容器中，元素访问时间与容器中元素个数的对数成正比。通常，这些 `hash` 容器还提供与标准容器相同的 `public` 函数，因此更容易使用。

使用遵循标准的容器时，代码将更容易在不同平台和编译器之间移植。另外，虽然遵循标准的容器的性能呈对数降低，但这可能并不会严重影响应用程序。

16.1.3 选择正确的容器

显然，可能有多种 STL 容器能够满足应用程序的需求，这时必须做出选择，这种选择很重要，因为错误的选择将导致应用程序的性能降低。

因此，在选择容器前，评估各种容器的优缺点很重要，如表 16.1 所示。

表 16.1 STL 容器类的特点

容器	类型	优点	缺点
<code>std::vector</code>	顺序容器	在末尾插入数据时快（时间固定） 可以像访问数组一样进行访问	调整大小时将影响性能 搜索时间与容器中的元素个数成正比 只能在末尾插入数据
<code>std::deque</code>	顺序容器	具备 <code>vector</code> 的所有优点，还可在容器开头插入数据，插入时间也是固定的	有 <code>vector</code> 的所有缺点 但与 <code>vector</code> 不同的是，根据规范， <code>deque</code> 不需要支持 <code>reserve()</code> 函数，该函数让程序员能够给 <code>vector</code> 预留内存空间，以免频繁地调整大小以提高性能
<code>std::list</code>	顺序容器	在 <code>list</code> 开头、中间或末尾插入数据，所需时间都是固定的 将元素从 <code>list</code> 中删除所需的时间是固定的，而不管元素的位置如何 插入或删除元素后，指向 <code>list</code> 中其他元素的迭代器仍有效	不能像数组那样根据索引随机访问元素 搜索速度比 <code>vector</code> 慢，因为元素没有存储在连续的内存单元中 搜索时间与容器中的元素个数成正比
<code>std::set</code>	关联容器	搜索时间不与容器中的元素个数成正比，因此搜索速度通常比顺序容器快得多	元素的插入速度比顺序容器慢
<code>std::multiset</code>	关联容器	优点与 <code>std::set</code> 类似 需要在排序的容器中存储非唯一的元素时，可使用这种容器	缺点与 <code>std::set</code> 类似
<code>std::map</code>	关联容器	用于存储键-值对的容器，并根据键进行排序 搜索时间不与容器中的元素个数成正比，因此搜索速度通常比顺序容器快得多	
<code>std::multimap</code>	关联容器	优点与 <code>std::map</code> 类似 在需要存储键-值且要求键不唯一时，可选择这种容器	缺点与 <code>std::map</code> 类似

16.2 STL 迭代器

最简单的迭代器是指针。给定一个指向数组中的第一个元素的指针，可递增该指针使其指向下一

个元素，还可直接对当前位置的元素进行操作。

STL 中的迭代器是模板类，从某种程度上说，它们是泛型指针。这些模板类让程序员能够对 STL 容器进行操作。注意，操作也可以是以模板函数的方式提供的 STL 算法，迭代器是一座桥梁，让这些模板函数能够以一致而无缝的方式处理容器，而容器是模板类。

STL 提供的迭代器分两大类。

- **输入迭代器**：通过对输入迭代器解除引用，它将引用对象，而对象可能位于集合中。最严格的输入迭代器确保只能以只读的方式访问对象。

- **输出迭代器**：输出迭代器让程序员对集合执行写入操作。最严格的输出迭代器确保只能对对象执行写入操作。

上述两种基本迭代器可进一步分为三类。

- **前向迭代器**：这是输入迭代器和输出迭代器的一种细化，它允许输入与输出。前向迭代器可以是 const 的，只能读取它指向的对象；也可以是可改变对象的，即可读写对象。前向迭代器通常应用于单向链表。

- **双向迭代器**：这是前向迭代器的一种细化，可对其执行递减操作，从而向后移动。双向迭代器通常用于双向链表。

- **随机访问迭代器**：这是对双向迭代器的一种细化，可将其加减一个偏移量，还可将两个迭代器相减以得到集合中两个元素之间的相对距离。随机访问迭代器通常用于数组。

#### 注意

从实现层面说，可将“细化”视为继承或具体化。

## 16.3 STL 算法

查找、排序和反转等都是标准的编程需求，不应让程序员重复实现这样的功能。因此 STL 以 STL 算法的方式提供这些函数，通过结合使用这些函数和迭代器，程序员可对容器执行一些最常见的操作。

最常用的 STL 算法包括：

- **std::find**，在集合中查找值；
- **std::find\_if**，根据用户指定的谓词在集合中查找值；
- **std::reverse**，反转集合中元素的排列顺序；
- **std::remove\_if**，根据用户定义的谓词将元素从集合中删除；
- **std::transform**，使用用户定义的变换函数对容器中的元素进行变换。

这些算法都是 std 命名空间中的模板函数，要使用它们，必须包含标准头文件 <algorithm>。

## 16.4 使用迭代器在容器和算法之间交互

下面通过一个示例阐述迭代器如何无缝地将容器和 STL 算法连接起来。程序清单 16.1 所示的程序使用了 STL 顺序容器 std::vector，该容器与动态数组类似。这个数组用于存储整数，程序使用 std::find 算法在集合中查找一个整数。请注意迭代器是如何在算法和其操作的容器之间搭建桥梁的。

程序清单 16.1 在 vector 中查找元素及其位置

```
1: #include <iostream>
2: #include <vector>
3: #include <algorithm>
4: using namespace std;
5:
```

```
6: int main ()
7: {
8:     // A dynamic array of integers
9:     vector<int> vecIntegerArray;
10:
11:     // Insert sample integers into the array
12:     vecIntegerArray.push_back (50);
13:     vecIntegerArray.push_back (2991);
14:     vecIntegerArray.push_back (23);
15:     vecIntegerArray.push_back (9999);
16:
17:     cout << "The contents of the vector are: " << endl;
18:
19:     // Walk the vector and read values using an iterator
20:     vector<int>::iterator iArrayWalker = vecIntegerArray.begin ();
21:
22:     while (iArrayWalker != vecIntegerArray.end ())
23:     {
24:         // Write the value to the screen
25:         cout << *iArrayWalker << endl;
26:
27:         // Increment the iterator to access the next element
28:         ++ iArrayWalker;
29:     }
30:
31:     // Find an element (say 2991) in the array using the 'find' algorithm...
32:     vector<int>::iterator iElement = find (vecIntegerArray.begin ()
33:                                         ,vecIntegerArray.end (), 2991);
34:
35:     // Check if value was found
36:     if (iElement != vecIntegerArray.end ())
37:     {
38:         // Value was found... Determine position in the array:
39:         int nPosition = distance (vecIntegerArray.begin (), iElement);
40:         cout << "Value " << *iElement;
41:         cout << " found in the vector at position: " << nPosition << endl;
42:     }
43:
44:     return 0;
45: }
```

#### ▼ 输出:

```
The contents of the vector are:
50
2991
23
9999
Value 2991 found in the vector at position: 1
```

#### ▼ 分析:

程序清单 16.1 演示了如何使用迭代器遍历向量。迭代器是一个接口，将算法（find）连接到其要操作的数据所属的容器（如 vector）。第 20 行声明了迭代器对象 iArrayWalker，并将其初始化为指向容器开头，即 vector 的成员函数 begin() 返回的值。第 22~29 行演示了如何在循环中使用该迭代器遍历并显示 vector 包含的元素，这与显示静态数组的内容极其相似。迭代器的用法在所有 STL 容器中都相同。所有容器都提供了 begin() 函数和 end() 函数，其中前者指向第一个元素，后者指向容器中最后一个元素的后面。这就是第 22 行的 while 循环在 end() 前面而不是 end() 处结束的原因。第 32 行演示了如何使用 find 在 vector 中查找值。find 操作的结果也是一个迭代器，通过将该迭代器与容器末尾进行比较，可判断 find 是否成功，如第 36 行所示。如果找到了元素，便可对该迭代器解除引用（就像对指针解除引用一样）以显示该元素。算法 distance 计算找到的元素的所处位置的偏移量。

#### 注意

第 18 章将详细介绍 std::vector，第 23 章将详细介绍 STL 算法。

## 16.5 总结

本章介绍了 STL 容器、迭代器和算法后面的基本概念，它们可能是最重要的 STL 元素，深入理解它们背后的概念有助于在应用程序中高效地使用 STL。第 17~25 章将更详细地解释这些概念的实现及其应用。

## 16.6 问与答

问：我需要一个数组，但不知道它应包含多少个元素。请问我应使用哪种 STL 容器？

答：std::vector 或 std::deque 能够很好地满足这种需求。这两种容器都负责管理内存，并可根据应用程序需求动态地调整大小。

问：我的应用程序经常需要执行搜索操作，我应选择哪种容器？

答：关联容器最适合于需要经常进行搜索的应用程序。

问：我要存储键-值对，并希望能够快速完成查询，但键可能不是唯一的。我应选择哪种容器？

答：std::multimap 类型的关联容器适合这种需求。multimap 可存储非唯一的键-值对，查询速度也快，这是关联容器的一个特点。

问：我要开发一个能够在不同平台和编译器之间移植的应用程序，该程序还需要使用能够根据键快速查询的容器。我应使用 std::map 还是 std::hash\_map？

答：移植性是一个重要约束条件，必须使用遵循标准的容器，因此应选择 std::map。

## 16.7 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 测验

1. 要包含一个对象数组，并允许在开头和末尾插入对象，应使用哪种容器？
2. 要存储元素以进行快速查找，应选择哪种容器？
3. 要使用 std::set 存储元素，并根据除元素值外的其他条件进行存储和查找，可能吗？
4. STL 的哪部分将算法和容器连接起来，让算法能够操作容器中的元素？
5. 如果应用程序要移植到不同的平台，并使用不同的 C++ 编译器进行编译，是否可选择使用容器 hash\_set？

## 第 17 章

# STL string 类

标准模板库 (STL) 向程序员提供了一个用于字符串操作的容器类。string 类不仅能够根据应用程序的需求动态调整其大小，还提供了很有用的助手函数可帮助操作字符串，这让程序员能够在应用程序中使用标准的、经过测试的可移植功能，并将其主要精力放在开发应用程序的重要功能上。

在本章中，您将学习：

- 为何需要字符串操作类
- 使用 STL string 类
- 基于模板的 STL string 实现

### 17.1 为何需要字符串操作类

在 C++ 中，字符串是一个字符数组。第 3 章介绍过，最简单的字符数组可这样定义：

```
char pszName [20];
```

该语句声明了一个包含 20 个元素的字符数组。可以看到，这个缓冲区可存储一个长度有限的字符串，如果试图存储的字符数超出限制将溢出。不能调整静态数组的长度，为避开这种限制，C++ 支持动态分配内存，因此可以如下定义更动态的字符数组：

```
char* pszName = new char [nArrayLength];
```

这定义了一个动态分配的字符数组，其长度由变量 nArrayLength 的值指定，而这种值是在运行阶段确定的，因此该数组的长度是可变的。然而，如果要在运行阶段改变数组的长度，必须首先释放以前分配给它的内存，再重新分配内存来存储数据。

如果以字符数组的方式实现的字符串是类的成员属性，情况将更复杂。将对象赋值给另一个对象时，如果编写的复制构造函数和赋值运算符不完善，可能导致复制对象时复制成员字符串的地址。当两个对象的字符串指针指向相同的内存地址时，如果源对象被销毁，目标对象中的指针将非法。

为解决这种问题，可使用字符串类，而不使用字符数组或字符指针。STL string 类 std::string 可提供如下帮助：

- 减少程序员在创建和操作字符串方面所需做的工作；
  - 提供在内部管理内存分配细节的应用程序的稳定性；
  - 提供可确保成员字符串得以正确的复制构造函数和赋值运算符；
  - 提供有助于复制、截短、查找和删除等操作的实用函数；
  - 提供用于比较的运算符；
  - 让程序员能够将精力放在应用程序的主要需求而不是字符串操作细节上。
- 稍后将介绍 string 类提供的一些助手函数。



## 17.2 使用 STL string 类

最常用的字符串函数包括：

- 复制；
- 连接；
- 查找字符和子字符串；
- 截短；
- 使用标准模板库提供的算法实现字符串反转和大小写转换。

要使用 STL string 类，必须包含头文件<string>。

### 17.2.1 实例化 STL string 及复制

string 类提供了很多重载的构造函数，因此可以多种方式进行实例化和初始化。例如，初始化一个常量字符串并将其赋给一个 STL string 对象：

```
const char* pszConstString = "Hello String!";
std::string strFromConst (pszConstString);
```

或：

```
std::string strFromConst = pszConstString;
```

上述代码与下面的代码类似：

```
std::string str2 ("Hello String!");
```

显然，实例化并初始化 string 对象时，无需关心字符串长度和内存分配细节。STL string 类的构造函数将自动完成这些工作。

同样，可使用一个 string 对象来初始化另一个：

```
std::string str2Copy (str2);
```

可让 string 的构造函数只接受输入字符串的前 n 个字符：

```
// Initialize a string to the first 5 characters of another
std::string strPartialCopy (pszConstString, 5);
```

还可这样初始化 string 对象，即使其包含指定数量的特定字符：

```
// Initialize a string object to contain 10 'a's
std::string strRepeatChars (10, 'a');
```

程序清单 17.1 演示了实例化和复制 STL string 的方法。

#### 程序清单 17.1 实例化和复制 STL string 的方法

```
1: #include <string>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:     const char* pszConstString = "Hello String!";
8:     cout << "Constant string is: " << pszConstString << endl;
9:
10:    std::string strFromConst (pszConstString);
11:    cout << "strFromConst is: " << strFromConst << endl;
12:
13:    std::string str2 ("Hello String!");
14:    std::string str2Copy (str2);
15:    cout << "str2Copy is: " << str2Copy << endl;
16:
17:    // Initialize a string to the first 5 characters of another
18:    std::string strPartialCopy (pszConstString, 5);
19:    cout << "strPartialCopy is: " << strPartialCopy << endl;
20:
21:    // Initialize a string object to contain 10 'a's
22:    std::string strRepeatChars (10, 'a');
```

```

23:     cout << "strRepeatChars is: " << strRepeatChars << endl;
24:
25:     return 0;
26: }

```

### ▼ 输出:

```

Constant string is: Hello String!
strFromConst is: Hello String!
str2Copy is: Hello String!
strPartialCopy is: Hello
strRepeatChars is: aaaaaaaaaa

```

### ▼ 分析:

上述示例代码演示如何实例化一个 STL string 对象, 并将其初始化为另一个字符串, pszConstString 是一个包含示例值的 C 风格字符串, 它是第 7 行初始化的。从第 10 行可知, 使用 std::string 的构造函数进行复制非常简单。第 13 行将另一个常量字符串复制给 std::string 对象 str2, 第 14 行演示了如何使用 std::string 的另一个重载构造函数来复制 std::string 对象, 从而获得 str2Copy。第 18 行演示了如何进行部分复制。第 22 行演示如何实例化一个 std::string 对象, 并将其初始化为包含多个相同的字符。这段代码只是小型演示程序, 它演示了通过使用 std::string 及其众多复制构造函数, 创建、复制和显示字符串很容易。

注意, 如果要将一个 C 风格字符串复制到另一个 C 风格字符串中, 第 10 行代码将为:

```

const char* pszConstString = "Hello World!"; // To be copied

// To create a copy, first allocate memory for one...
char * pszCopy = new char [strlen (pszConstString) + 1];
strcpy (pszCopy, pszConstString); // The copy step

// deallocate memory after using pszCopy
delete [] pszCopy;

```

可以看到, 这需要更多的代码, 导致错误的可能性也更大, 同时程序员还需要负责管理内存的分配与释放。这些工作 STL string 都能完成, 还能完成其他工作!

## 17.2.2 访问 string 及其内容

要访问 STL string 的字符内容, 可使用迭代器, 也可采用类似于数组的语法并使用下标运算符 ([]) 提供偏移量。要获得 string 对象的 C 风格表示, 可使用成员函数 c\_str(), 如程序清单 17.2 所示。

程序清单 17.2 访问 STL string 的字符元素

```

1: #include <string>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // The sample string
9:     string strSTLString ("Hello String");
10:
11:     // Access the contents of the string using array syntax
12:     cout << "Displaying characters using array-syntax: " << endl;
13:     for ( size_t nCharCounter = 0
14:           ; nCharCounter < strSTLString.length ()
15:           ; ++ nCharCounter )
16:     {
17:         cout << "Character [" << nCharCounter << "] is: ";
18:         cout << strSTLString [nCharCounter] << endl;
19:     }
20:     cout << endl;
21: }

```

```

22:    // Access the contents of a string using iterators
23:    cout << "Displaying characters using iterators: " << endl;
24:    int nCharOffset = 0;
25:    string::const_iterator iCharacterLocator;
26:    for ( iCharacterLocator = strSTLString.begin ()
27:          ; iCharacterLocator != strSTLString.end ()
28:          ; ++ iCharacterLocator )
29:    {
30:        cout << "Character [" << nCharOffset ++ << "] is: ";
31:        cout << *iCharacterLocator << endl;
32:    }
33:    cout << endl;
34:
35:    // Access the contents of a string as a C-style string
36:    cout << "The char* representation of the string is: ";
37:    cout << strSTLString.c_str () << endl;
38:
39:    return 0;
40: }

```

### ▼ 输出:

Displaying the elements in the string using array-syntax:

```

Character [0] is: H
Character [1] is: e
Character [2] is: l
Character [3] is: l
Character [4] is: o
Character [5] is:
Character [6] is: S
Character [7] is: t
Character [8] is: r
Character [9] is: i
Character [10] is: n
Character [11] is: g

```

Displaying the contents of the string using iterators:

```

Character [0] is: H
Character [1] is: e
Character [2] is: l
Character [3] is: l
Character [4] is: o
Character [5] is:
Character [6] is: S
Character [7] is: t
Character [8] is: r
Character [9] is: i
Character [10] is: n
Character [11] is: g

```

The char\* representation of the string is: Hello String

### ▼ 分析:

上述代码演示了访问 string 内容的多种方法。迭代器很重要，因为很多 string 成员函数都以迭代器的方式返回其结果。第 12~19 行使用 std::string 类实现的下标运算符（[]）以类似数组的语法显示 string 中的字符。注意，这个运算符要求提供偏移量，如第 18 行所示。因此，确保不超出 string 的边界很重要，即读取字符时，提供的偏移量不能大于 string 的长度。第 26~32 行也逐字符显示 string 的内容，但使用的是迭代器。

## 17.2.3 字符串连接

要连接字符串，可使用运算符+=，也可使用成员函数 append，如程序清单 17.3 所示。

### 程序清单 17.3 使用 STL string 连接字符串

```

1: #include <string>
2: #include <iostream>
3:

```

```
4: int main ()
5: {
6:     using namespace std;
7:
8:     string strSample1 ("Hello");
9:     string strSample2 (" String!");
10:
11:     // Concatenate
12:     strSample1 += strSample2;
13:     cout << strSample1 << endl << endl;
14:
15:     string strSample3 (" Fun is not needing to use pointers!");
16:     strSample1.append (strSample3);
17:     cout << strSample1 << endl << endl;
18:
19:     const char* pszConstString = "You however still can!";
20:     strSample1.append (pszConstString);
21:     cout << strSample1 << endl;
22:
23:     return 0;
24: }
```

#### ▼ 输出:

```
Hello String!
Hello String! Fun is not needing to use pointers!
```

```
Hello String! Fun is not needing to use pointers when working with strings. You
however still can!
```

#### ▼ 分析:

第 12、16 和 20 行演示了连接 STL string 的各种方法。请注意运算符+=的用法以及 append 函数的功能。append 函数有多个重载版本，能够接受另一个 string 对象（如第 11 行所示），还能接受一个 C 风格字符串。

### 17.2.4 在 string 中查找字符或子字符串

STL string 类提供了成员函数 find，该函数有多个重载版本，可在给定 string 对象中查找字符或子字符串。程序清单 17.4 演示了这个成员函数的用法。

程序清单 17.4 使用函数 string::find

```
1: #include <string>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     string strSample ("Good day String! Today is beautiful!");
9:     cout << "The sample string is: " << endl;
10:    cout << strSample << endl << endl;
11:
12:    // Find substring "day" in it...
13:    size_t nOffset = strSample.find ("day", 0);
14:
15:    // Check if the substring was found...
16:    if (nOffset != string::npos)
17:        cout << "First instance of \"day\" was found at offset " << nOffset;
18:    else
19:        cout << "Substring not found." << endl;
20:
21:    cout << endl << endl;
22:
23:    cout << "Locating all instances of substring \"day\"" << endl;
24:    size_t nSubstringOffset = strSample.find ("day", 0);
```

```

25:
26:     while (nSubstringOffset != string::npos)
27:     {
28:         cout << "\"day\" found at offset " << nSubstringOffset << endl;
29:
30:         // Make the 'find' function search the next character onwards
31:         size_t nSearchOffset = nSubstringOffset + 1;
32:
33:         nSubstringOffset = strSample.find("day", nSearchOffset);
34:     }
35:
36:     cout << endl;
37:
38:     cout << "Locating all instances of character 'a' " << endl;
39:     const char chCharToSearch = 'a';
40:     size_t nCharacterOffset = strSample.find(chCharToSearch, 0);
41:
42:     while (nCharacterOffset != string::npos)
43:     {
44:         cout << "'" << chCharToSearch << " found";
45:         cout << " at position: " << nCharacterOffset << endl;
46:
47:         // Make the 'find' function search forward from the next character
onwards
48:         size_t nCharSearchOffset = nCharacterOffset + 1;
49:
50:         nCharacterOffset = strSample.find(chCharToSearch, nCharSearchOffset);
51:     }
52:
53:     return 0;
54: }

```

**▼ 输出:**

```

The sample string is:
Good day String! Today is beautiful!

First instance of "day" was found at offset 5

Locating all instances of substring "day"
"day" found at offset 5
"day" found at offset 19

Locating all instances of character 'a'
'a' found at position: 6
'a' found at position: 20
'a' found at position: 28

```

**▼ 分析:**

第 13~19 行演示了 find 函数的最简单用法，它判断是否在 string 中找到了特定的子字符串。这是通过将 find 操作的结果与 std::string::npos（实际值为-1）进行比较实现的，std::string::npos 表明没有找到要搜索的元素。如果 find 函数没有返回 npos，它将返回一个偏移量，指出子字符串或字符在 string 中的位置。

这段代码演示了如何在 while 循环中使用 find 函数在 STL string 查找指定字符或子字符串的所有实例。这里使用的 find 函数的重载版本接受两个参数：要搜索的子字符串或字符以及命令 find 从哪里开始搜索的偏移量。

**注意**

STL string 还有一些与 find 类似的函数，如 find\_first\_of、find\_first\_not\_of、find\_last\_of 和 find\_last\_not\_of，这些函数可满足程序员的其他编程需求。

### 17.2.5 截短 STL string

STL string 类提供了 erase 函数，可用于：



- 在给定偏移位置和字符数时删除指定数目的字符；
- 在给定指向字符的迭代器时删除字符；
- 在给定由两个迭代器指定的范围时删除该范围内的字符。

程序清单 17.5 中的示例演示了 `string::erase()` 函数的各种重载版本的用途。

程序清单 17.5 使用成员函数 `erase` 截短字符串

```
1: #include <string>
2: #include <algorithm>
3: #include <iostream>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     string strSample ("Hello String! Wake up to a beautiful day!");
10:    cout << "The original sample string is: " << endl;
11:    cout << strSample << endl << endl;
12:
13:    // Delete characters from the string given position and count
14:    cout << "Truncating the second sentence: " << endl;
15:    strSample.erase (13, 28);
16:    cout << strSample << endl << endl;
17:
18:    // Find a character 'S' in the string using STL find algorithm
19:    string::iterator iCharS = find ( strSample.begin ()
20:                                     , strSample.end (), 'S');
21:
22:    // If character found, 'erase' to deletes a character
23:    cout << "Erasing character 'S' from the sample string:" << endl;
24:    if (iCharS != strSample.end ())
25:        strSample.erase (iCharS);
26:
27:    cout << strSample << endl << endl;
28:
29:    // Erase a range of characters using an overloaded version of erase()
30:    cout << "Erasing a range between begin() and end(): " << endl;
31:    strSample.erase (strSample.begin (), strSample.end ());
32:
33:    // Verify the length after the erase() operation above
34:    if (strSample.length () == 0)
35:        cout << "The string is empty" << endl;
36:
37:    return 0;
38: }
```

#### ▼ 输出:

```
The original sample string is:
Hello String! Wake up to a beautiful day!

Truncating the second sentence:
Hello String!

Erasing character 'S' from the sample string:
Hello tring!

Erasing a range between begin() and end():
The string is empty
```

#### ▼ 分析:

该程序清单演示 `erase` 函数的 3 个版本。其中一个版本在给定偏移位置和字符数的情况下删除指定数目的字符，如第 15 行所示；另一个版本在给定指向字符的迭代器的情况下删除指定的字符，如第 25 行所示；最后一个版本在给定由两个迭代器指定的范围的情况下删除该范围内的字符，如第 31 行所示。在这里，范围是由 `string` 的成员函数 `begin()` 和 `end()` 指定的，它包含字符串的所有内容，因此对该范围调用 `erase()` 将清除 `string` 对象的全部内容。注意，`string` 类还提供了 `clear()` 函数，该函数清除全

部内容并重置 string 对象。

### 17.2.6 字符串反转

有时需要反转字符串的内容，如判断用户输入的字符串是否为回文时。反转 STL string 很容易，只需使用 `std::reverse` 算法，如程序清单 17.6 所示。

程序清单 17.6 反转 STL string

```
1: #include <string>
2: #include <iostream>
3: #include <algorithm>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     string strSample ("Hello String! We will reverse you!");
10:    cout << "The original sample string is: " << endl;
11:    cout << strSample << endl << endl;
12:
13:    reverse (strSample.begin (), strSample.end ());
14:
15:    cout << "After applying the std::reverse algorithm: " << endl;
16:    cout << strSample;
17:
18:    return 0;
19:
20: }
```

▼ 输出:

```
The original sample string is:
Hello String! We will reverse you!

After applying the std::reverse algorithm:
luoy esrever lliw eW !gnirtS olleH
```

▼ 分析:

第 13 行的 `std::reverse` 算法根据两个输入参数指定的边界反转边界内的内容。在这里，两个边界分别是 `string` 对象的开头和末尾，因此整个字符串都被反转。只要提供合适的输入参数，也可将字符串的一部分反转。注意，边界不能超过 `end()`。

### 17.2.7 字符串的大小写转换

要对字符串进行大小写转换，可使用算法 `std::transform`，它对集合中的每个元素执行一个用户指定的函数。在这里，集合是 `string` 对象本身。程序清单 17.7 演示了如何对 `string` 中的字符进行大小写转换。

程序清单 17.7 使用 `std::transform` 对 STL string 进行大小写转换

```
1: #include <string>
2: #include <iostream>
3: #include <algorithm>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     cout << "Please enter a string for case-conversion:" << endl;
10:    cout << "> ";
11:
12:    string strInput;
```

```

13:    getline (cin, strInput);
14:    cout << endl;
15:
16:    transform (strInput.begin(),strInput.end(),strInput.begin(),toupper);
17:    cout << "The string converted to upper case is: " << endl;
18:    cout << strInput << endl << endl;
19:
20:    transform (strInput.begin(),strInput.end(),strInput.begin(),tolower);
21:    cout << "The string converted to lower case is: " << endl;
22:    cout << strInput << endl << endl;
23:
24:    return 0;
25: }

```

#### ▼ 输出:

```

Please enter a string for case-conversion:
> Convert thIS StrIng!

```

```

The string converted to upper case is:
CONVERT THIS STRING!

```

```

The string converted to lower case is:
convert this string!

```

#### ▼ 分析:

第 12 行和第 15 行演示了如何使用 `std::transform` 来改变 STL string 中字符的大小写。

## 17.3 基于模板的 STL string 实现

前面说过，`std::string` 类实际上是 STL 模板类 `std::basic_string <T>` 的具体化。容器类 `basic_string` 的模板声明如下：

```

template<class _Elem,
        class _Traits,
        class _Ax>
class basic_string

```

在该模板定义中，最重要的参数是第一个：`_Elem`，它指定了 `basic_string` 对象将存储的数据类型。因此，`std::string` 使用 `_Elem=char` 具体化模板 `basic_string` 的结果，而 `wstring` 使用 `_Elem= wchar` 具体化模板 `basic_string` 的结果。

换句话说，STL string 类的定义如下：

```

typedef basic_string<char, char_traits<char>, allocator<char> >
string;

```

而 STL wstring 类的定义如下：

```

typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >
wstring;

```

因此，前面介绍的所有 string 功能和函数实际上都是 `basic_string` 提供的，它们也适用于 STL wstring 类。

## 17.4 总结

本章介绍了 STL string 类，它是标准模板库提供的一个容器，可满足程序员众多的字符串操作需求。使用这个类的优点是，原本由程序员负责的内存管理、字符串比较和字符串操作都将由 STL 框架提供的一个容器类完成。

## 17.5 问与答

问：我要使用 `std::reverse` 来反转一个字符串，要使用这个函数，需要包含哪个头文件？

答：要使用 `std::reverse`，需要包含头文件 `<algorithm>`。

问：在使用 `tolower()` 函数将字符串转换为小写时，`std::transform` 的作用是什么？

答：`std::transform` 对 `string` 对象中指定边界内的每个字符调用 `tolower()` 函数。

问：为什么 `std::wstring` 和 `std::string` 的行为和成员函数完全相同？

答：因为它们都是具体化模板类 `std::basic_string` 的结果。

问：STL `string` 类的比较运算符<在执行比较时是否区分大小写？

答：区分大小写。

## 17.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 17.6.1 测验

1. `std::string` 具体化了哪个 STL 模板类？
2. 如果要对两个字符串进行区分大小写的比较，该如何做？
3. STL `string` 与 C 风格字符串是否类似？

### 17.6.2 练习

1. 编写一个程序检查用户输入的单词是否为回文。例如，ATOYOTA 是回文，因为反转后单词与原来相同。
2. 编写一个程序，告诉用户输入的句子包含多少个元音字母。
3. 将字符串的字符交替地转换为大写。
4. 编写一个程序，它将 4 个 `string` 对象分别被初始化为 I、Love、STL 和 String.，然后在这些字符串之间添加空格，再显示整个句子。

资源分享  
PDG

## 第 18 章

# STL 动态数组类

动态数组让程序员能够灵活地存储数据，同时程序员无需像使用静态数组那样在编写应用程时就需要知道数组的长度。显然，这是一种常见的需求，标准模板库（STL）通过 `std::vector` 类提供了现成的解决方案。

在本章中，您将学习：

- `std::vector` 的特点
- 典型的 `vector` 操作
- 理解大小与容量的概念
- STL `deque` 类

### 18.1 `std::vector` 的特点

`vector` 是一个模板类，提供了动态数组的通用功能，具有如下特点：

- 在数组末尾添加元素所需的时间是固定的，即在末尾插入元素的所需时间不随数组大小而异，在末尾删除元素也如此；
- 在数组中间添加或删除元素所需的时间与该元素后面的元素个数成正比；
- 存储的元素数是动态的，而 `vector` 类负责管理内存。

`vector` 是一种动态数组，其结构如图 18.1 所示。

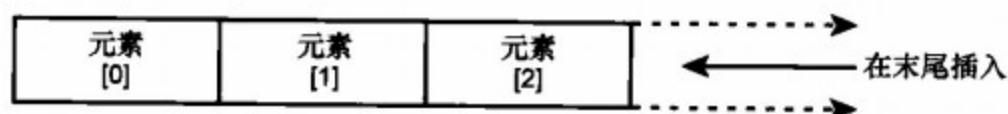


图 18.1 向量的内部构造

要使用 `std::vector` 类，需要包含下面的头文件：

```
#include <vector>
```

### 18.2 典型的 `vector` 操作

`std::vector` 类的行为规范和公有成员是由 C++ 标准定义的，因此，遵循该标准的所有 C++ 编程平台都支持本章将介绍的 `vector` 操作。

#### 18.2.1 实例化 `vector`

`vector` 是一个模板类，需要使用第 15 章介绍的方法进行实例化。要实例化 `vector`，需要指定要在动态数组中存储的对象类型。这看起来很复杂，但实际上非常简单，如程序清单 18.1 所示。



程序清单 18.1 实例化 std::vector

```
1: #include <vector>
2:
3: int main ()
4: {
5:     std::vector <int> vecDynamicIntegerArray;
6:
7:     // Instantiate a vector with 10 elements (it can grow larger)
8:     std::vector <int> vecArrayWithTenElements (10);
9:
10:    // Instantiate a vector with 10 elements, each initialized to 90
11:    std::vector <int> vecArrayWithTenInitializedElements (10, 90);
12:
13:    // Instantiate one vector and initialize it to the contents of another
14:    std::vector <int> vecArrayCopy (vecArrayWithTenInitializedElements);
15:
16:    // Instantiate a vector to 5 elements taken from another
17:    std::vector<int> vecSomeElementsCopied(vecArrayWithTenElements.begin()
18:                                           , vecArrayWithTenElements.begin () + 5);
19:
20:    return 0;
21: }
```

### ▼ 分析:

上述代码演示了如何为整型具体化 vector 类, 即实例化一个存储整型数据的 vector。该 vector 名为 vecDynamicIntegerArray, 它使用了默认构造函数。在不知道容器最小需要多大, 即不知道要存储多少个整数时, 默认构造函数很有用。实例化 vector 的第 2 种和第 3 种方式如第 11 行和第 14 行所示, 在这里, 程序员知道 vector 至少应包含 10 个元素。注意, 这并没有限制容器最终的大小, 而只是设置了初始大小。第 4 种形式如第 17 行和第 18 行所示, 它使用一个 vector 实例化另一个 vector 的内容, 即复制 vector 对象或其一部分。这是所有 STL 容器都支持的构造函数。最后一种形式是使用迭代器。vecSomeElementCopied 包含 vecArrayWithTenElements 的前 5 个元素。

### 注意

第 4 个构造函数只能用于类型类似的对象, 因此可使用一个包含整型对象的 vector 来实例化另一个整型 vector, 但如果其中一个 vector 包含的对象类型为 float, 代码将不能通过编译。

## 18.2.2 在 vector 中插入元素

实例化一个整型 vector 后, 接下来需要在 vector 中插入元素 (整数)。在 vector 中插入元素时, 元素将插入到数组末尾, 这是使用成员方法 push\_back 完成的, 如程序清单 18.2 所示。

程序清单 18.2 使用 push\_back 方法在 vector 中插入元素

```
1: #include <iostream>
2: #include <vector>
3:
4: int main ()
5: {
6:     std::vector <int> vecDynamicIntegerArray;
7:
8:     // Insert sample integers into the vector:
9:     vecDynamicIntegerArray.push_back (50);
10:    vecDynamicIntegerArray.push_back (1);
11:    vecDynamicIntegerArray.push_back (987);
12:    vecDynamicIntegerArray.push_back (1001);
13:
14:    std::cout << "The vector contains ";
15:    std::cout << vecDynamicIntegerArray.size () << " Elements";
16:
17:    return 0;
18: }
```

**▼ 输出:**

The vector contains 4 Elements

**▼ 分析:**

第 9~12 行中的 `push_back` 是 `vector` 类的一个公有成员方法, 用于在动态数组末尾插入对象。请注意函数 `size()` 的用法, 它返回 `vector` 中存储的元素个数。

另一个插入元素的方法是, 指定 `vector` 要存储的元素个数, 然后像使用数组那样将值分别复制到各个位置。

**程序清单 18.3 使用数组语法设置 vector 中的值**

```
1: #include <vector>
2: #include <iostream>
3:
4: int main ()
5: {
6:     std::vector<int> vecDynamicIntegerArray (4);
7:
8:     // Copy integer values into individual element locations
9:     vecDynamicIntegerArray [0] = 50;
10:    vecDynamicIntegerArray [1] = 1;
11:    vecDynamicIntegerArray [2] = 987;
12:    vecDynamicIntegerArray [3] = 1001;
13:
14:    std::cout << "The vector contains ";
15:    std::cout << vecDynamicIntegerArray.size () << " Elements";
16:
17:    return 0;
18: }
```

**▼ 输出:**

The vector contains 4 Elements

**▼ 分析:**

上述代码实例化了一个整型 `vector`, 并指定该 `vector` 最初包含的元素个数。第 6 行创建一个包含 4 个整数的 `vector` 对象, 接下来的几行使用数组语法将值复制到 `vector` 的 4 个位置。使用下标运算符 (`[]`) 时, 仅当偏移位置位于 `vector` 边界内时才是安全的。最后, 代码显示 `vector` 包含的元素个数。可以看到, 输出与程序清单 18.1 的输出相同, 而程序清单 18.1 使用 `push_back` 将元素插入到未初始化的 `vector` 中。

注意, 将 `vector` 初始化为包含 4 个元素并没有限制它只能包含这么多元素, 程序员可根据应用程序的需求使用 `push_back` 将元素插入到 `vector` 中。

除提供了 `push_back` 以及允许使用下标运算符将值存储到已初始化的 `vector` 中外, 与很多 STL 容器一样, `std::vector` 还提供了 `insert` 函数。使用 `insert` 函数时, 需要指定要将元素插入到什么位置, 如程序清单 18.4 所示。

**程序清单 18.4 使用 `vector::insert` 在中间插入元素**

```
1: #include <vector>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // Instantiate a vector with 4 elements, each initialized to 90
9:     vector<int> vecIntegers (4, 90);
10:
11:     cout << "The initial contents of the vector are: ";
12:
13:     vector<int>::iterator iElement;
14:     for ( iElement = vecIntegers.begin ()
```

```

15:         ; iElement != vecIntegers.end ()
16:         ; ++ iElement )
17:     {
18:         cout << *iElement << ' ';
19:     }
20:
21:     cout << endl;
22:
23:     // Insert 25 at the beginning
24:     vecIntegers.insert (vecIntegers.begin (), 25);
25:
26:     cout << "The vector after inserting an element at the beginning: ";
27:     for ( iElement = vecIntegers.begin ()
28:           ; iElement != vecIntegers.end ()
29:           ; ++ iElement )
30:     {
31:         cout << *iElement << ' ';
32:     }
33:
34:     cout << endl;
35:
36:     // Insert 2 numbers of value 45 at the end
37:     vecIntegers.insert (vecIntegers.end (), 2, 45);
38:
39:     cout << "The vector after inserting two elements at the end: ";
40:     for ( iElement = vecIntegers.begin ()
41:           ; iElement != vecIntegers.end ()
42:           ; ++ iElement )
43:     {
44:         cout << *iElement << ' ';
45:     }
46:     cout << endl;
47:
48:     // Another vector containing 2 elements of value 30
49:     vector <int> vecAnother (2, 30);
50:
51:     // Insert two elements from another container in position [1]
52:     vecIntegers.insert (vecIntegers.begin () + 1,
53:                         vecAnother.begin (), vecAnother.end ());
54:
55:     cout << "The vector after inserting contents from another ";
56:     cout << "in the middle:" << endl;
57:     for ( iElement = vecIntegers.begin ()
58:           ; iElement != vecIntegers.end ()
59:           ; ++ iElement )
60:     {
61:         cout << *iElement << ' ';
62:     }
63:
64:     return 0;
65: }

```

### ▼ 输出:

```

The initial contents of the vector are: 90 90 90 90
The vector after inserting an element at the beginning: 25 90 90 90 90
The vector after inserting two elements at the end: 25 90 90 90 90 45 45
The vector after inserting contents from another container in the middle:

25 30 30 90 90 90 90 45 45

```

### ▼ 分析:

上述代码演示了 insert 函数的强大功能, 它让您能够将值插入到容器中间。首先, 第 9 行创建了一个包含 4 个元素的 vector, 并将每个元素都初始化为 90; 然后, 使用成员函数 vector::insert 的各种重载版本插入元素, 如第 24、37 和 52 行所示, 它们分别演示了如何将值插入到 vector 的开头、中间和末尾。

### 注意

这可能是效率最低的将元素插入 vector 的方法 (当插入位置不是末尾时), 因为在开头或中间插入元素时, 将导致 vector 类将后面的所有元素后移 (为要插入的元素腾出空间)。

根据容器中包含的对象类型，这种移动操作可能需要调用复制构造函数或赋值运算符，因此开销可能很大。在上述例子中，vector 包含的是 int 对象，移动开销不是很大。但在其他情况下，情况可能并非如此。

### 18.2.3 访问 vector 中的元素

可使用下列方法访问 vector 的元素：使用下标运算符（[]）以数组语法方式访问；使用成员函数 at()；使用迭代器。

程序清单 18.5 演示如何使用下标运算符（[]）访问元素，而程序清单 18.3 使用它设置 vector 中的值。

程序清单 18.5 使用数组语法访问 vector 中的元素

```
1: #include <iostream>
2: #include <vector>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     vector <int> vecDynamicIntegerArray;
9:
10:    // Insert sample integers into the vector:
11:    vecDynamicIntegerArray.push_back (50);
12:    vecDynamicIntegerArray.push_back (1);
13:    vecDynamicIntegerArray.push_back (987);
14:    vecDynamicIntegerArray.push_back (1001);
15:
16:    unsigned int nElementIndex = 0;
17:    while (nElementIndex < vecDynamicIntegerArray.size ())
18:    {
19:        cout << "Element at position " << nElementIndex;
20:        cout << " is: " << vecDynamicIntegerArray [nElementIndex] << endl;
21:
22:        ++ nElementIndex;
23:    }
24:
25:    return 0;
26: }
```

#### ▼ 输出:

```
Element at position 0 is: 50
Element at position 1 is: 1
Element at position 2 is: 987
Element at position 3 is: 1001
```

#### ▼ 分析:

第 20 行表明，可以像静态数组那样，使用下标运算符（[]）访问 vector 的值。下标运算符接受一个从零开始的元素索引，与静态数组一样。

#### 警告

使用[]访问 vector 的元素时，面临的风险与访问数组元素相同，即不能超出容器的边界。使用下标运算符（[]）访问 vector 的元素时，如果指定的位置超出边界，结果将是不确定的（什么情况都可能发生，很可能是访问违规）。

更安全的方法是使用成员函数 at():

```
// gets element at position 2
cout << vecDynamicIntegerArray.at (2);
// the vector::at() version of the code above in Listing
➤18.5, line 20:
cout << vecDynamicIntegerArray.at (nElementIndex);
```

at()函数在运行阶段检查容器的大小，如果索引超出边界将引发异常。

注意，下标运算符（[]）只有在保证边界完整性的情况下才是安全的，如前一个例子所示。也可使用迭代器以指针语法访问 vector 中的元素，如程序清单 18.6 所示。

程序清单 18.6 使用指针语法（迭代器）访问元素

---

```

1: #include <iostream>
2: #include <vector>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     vector <int> vecDynamicIntegerArray;
9:
10:    // Insert sample integers into the vector:
11:    vecDynamicIntegerArray.push_back (50);
12:    vecDynamicIntegerArray.push_back (1);
13:    vecDynamicIntegerArray.push_back (987);
14:    vecDynamicIntegerArray.push_back (1001);
15:
16:    // Access objects in a vector using iterators:
17:    vector<int>::iterator iElementLocator = vecDynamicIntegerArray.begin();
18:
19:    while (iElementLocator != vecDynamicIntegerArray.end ())
20:    {
21:        size_t nElementIndex = distance (vecDynamicIntegerArray.begin (),
22:                                           iElementLocator);
23:
24:        cout << "Element at position ";
25:        cout << nElementIndex << " is: " << *iElementLocator << endl;
26:
27:        // move to the next element
28:        ++ iElementLocator;
29:    }
30:
31:    return 0;
32: }
```

---

#### ▼ 输出:

---

```

Element at position 0 is: 50
Element at position 1 is: 1
Element at position 2 is: 987
Element at position 3 is: 1001
```

---

#### ▼ 分析:

在这个例子中，迭代器有点像指针，迭代器的用法很像指针算术运算，如第 28 行和第 29 行所示。第 21 行使用 `std::distance` 来计算元素的偏移量，其中第二个参数是指向该元素的迭代器，而第一个参数是 vector 开头——`begin()` 返回的迭代器。

## 18.2.4 删除 vector 中的元素

除支持使用 `push_back` 方法在末尾插入元素外，vector 还支持使用 `pop_back` 函数将末尾的元素删除。使用 `pop_back` 将元素从 vector 中删除所需的时间是固定的，即不随 vector 存储的元素个数而异。程序清单 18.7 演示了如何使用函数 `pop_back` 删除 vector 末尾的元素。

程序清单 18.7 使用 `pop_back` 删除最后一个元素

---

```

1: #include <iostream>
2: #include <vector>
3:
4: int main ()
5: {
6:     using namespace std;
```

---



```

7:
8:   vector<int> vecDynamicIntegerArray;
9:
10:  // Insert sample integers into the vector:
11:  vecDynamicIntegerArray.push_back (50);
12:  vecDynamicIntegerArray.push_back (1);
13:  vecDynamicIntegerArray.push_back (987);
14:  vecDynamicIntegerArray.push_back (1001);
15:
16:  cout << "The vector contains ";
17:  cout << vecDynamicIntegerArray.size ();
18:  cout << " elements before calling pop_back" << endl;
19:
20:  // Erase one element at the end
21:  vecDynamicIntegerArray.pop_back ();
22:
23:  cout << "The vector contains ";
24:  cout << vecDynamicIntegerArray.size ();
25:  cout << " elements after calling pop_back" << endl;
26:
27:  cout << "Enumerating items in the vector... " << endl;
28:
29:  unsigned int nElementIndex = 0;
30:  while (nElementIndex < vecDynamicIntegerArray.size ())
31:  {
32:      cout << "Element at position " << nElementIndex << " is: ";
33:      cout << vecDynamicIntegerArray [nElementIndex] << endl;
34:
35:      // move to the next element
36:      ++ nElementIndex;
37:  }
38:  return 0;
39: }

```

#### ▼ 输出:

```

The vector contains 4 elements before calling pop_back
The vector contains 3 elements after calling pop_back
Enumerating items in the vector...
Element at position 0 is: 50
Element at position 1 is: 1
Element at position 2 is: 987

```

#### ▼ 分析:

上述输出表明,第20行的 pop\_back 函数将 vector 的最后一个元素删除,从而减少了 vector 包含的元素数量。第24行再次调用 size(),以证明 vector 包含的元素减少了一个,如输出所示。

## 18.3 理解 size()和 capacity()

vector 的大小指的是 vector 实际存储的元素数,而 vector 的容量指的是在重新分配内存以存储更多元素前 vector 能够存储的元素数。因此,vector 的大小小于或等于容量。

如果 vector 需要频繁地给其内部动态数组重新分配内存,将对性能造成一定的影响。在很大程度上说,这种问题可以通过使用成员函数 reserve(number)来解决。reserve 函数的功能基本上是增加分配给内部数组的内存,以免频繁地重新分配内存。通过减少重新分配内存的次数,还可减少复制对象的时间,从而提高性能,这取决于存储在 vector 中的对象类型。程序清单 18.8 说明了大小和容量之间的区别。

#### 程序清单 18.8 演示 size()和 capacity()

```

1: #include <iostream>
2: #include <vector>
3:
4: int main ()
5: {

```

```

6:     using namespace std;
7:
8:     // Instantiate a vector object that holds 5 integers of default value
9:     vector<int> vecDynamicIntegerArray (5);
10:
11:     cout << "Vector of integers was instantiated with " << endl;
12:     cout << "Size: " << vecDynamicIntegerArray.size ();
13:     cout << ", Capacity: " << vecDynamicIntegerArray.capacity () << endl;
14:
15:     // Inserting a 6th element in to the vector
16:     vecDynamicIntegerArray.push_back (666);
17:
18:     cout << "After inserting an additional element... " << endl;
19:     cout << "Size: " << vecDynamicIntegerArray.size ();
20:     cout << ", Capacity: " << vecDynamicIntegerArray.capacity () << endl;
21:
22:     // Inserting another element
23:     vecDynamicIntegerArray.push_back (777);
24:
25:     cout << "After inserting yet another element... " << endl;
26:     cout << "Size: " << vecDynamicIntegerArray.size ();
27:     cout << ", Capacity: " << vecDynamicIntegerArray.capacity () << endl;
28:
29:     return 0;
30: }

```

### ▼ 输出:

```

Vector of integers was instantiated with
Size: 5, Capacity: 5
After inserting an additional element...
Size: 6, Capacity: 7
After inserting yet another element...
Size: 7, Capacity: 7

```

### ▼ 分析:

第 9 行实例化了一个包含 5 个整型对象的 vector，这些整型对象使用默认值 0。第 12 行和第 13 行分别显示 vector 的大小和容量，它们在实例化后相等。第 16 行在 vector 中插入了第 6 个元素。鉴于在插入前 vector 的容量为 5，因此 vector 的内部缓冲区没有足够的内存来存储第 6 个元素。换句话说，vector 为扩大其容量以存储 6 个元素，需要重新分配内部缓冲区。重新分配的逻辑实现是智能的：为避免插入下一个元素时再次重新分配，提前分配了比当前需求更大的容量。

从输出可知，在容量为 5 的 vector 中插入第 6 个元素时，将容量增大到 7。size() 总是指出 vector 存储的元素数，当前其值为 6。第 23 行插入了第 7 个元素，这次没有扩大容量，因为已分配的内存足以满足需求。这时大小和容量相等，这表明 vector 的容量已经用完，再次插入元素将导致 vector 重新分配其内部缓冲区、复制现有的元素再插入新值。

### 注意

在重新分配 vector 内部缓冲区时提前增加容量方面，C++ 标准没有做任何规定，这取决于使用的 STL 实现。

## 18.4 STL deque 类

deque 是一个 STL 动态数组类，它与 vector 非常类似，但支持在数组开头和末尾插入或删除元素。deque 的内部结构如图 18.2 所示。

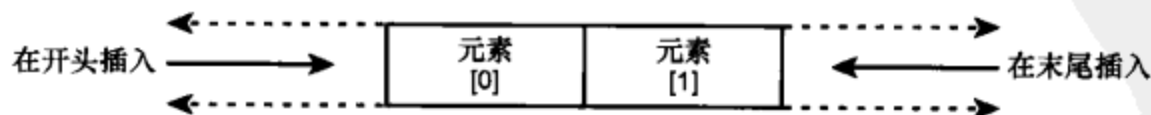


图 18.2 deque 的内部结构

要使用 deque 类, 需要包含头文件 <deque>。deque 的用法与 std::vector 极其相似, 如程序清单 18.9 所示。

程序清单 18.9 使用 STL deque 类

---

```

1: #include <deque>
2: #include <iostream>
3: #include <algorithm>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     // Define a deque of integers
10:    deque <int> dqIntegers;
11:
12:    // Insert integers at the bottom of the array
13:    dqIntegers.push_back (3);
14:    dqIntegers.push_back (4);
15:    dqIntegers.push_back (5);
16:
17:    // Insert integers at the top of the array
18:    dqIntegers.push_front (2);
19:    dqIntegers.push_front (1);
20:    dqIntegers.push_front (0);
21:
22:    cout << "The contents of the deque after inserting elements ";
23:    cout << "at the top and bottom are:" << endl;
24:
25:    // Display contents on the screen
26:    for ( size_t nCount = 0
27:        ; nCount < dqIntegers.size ()
28:        ; ++ nCount )
29:    {
30:        cout << "Element [" << nCount << "] = ";
31:        cout << dqIntegers [nCount] << endl;
32:    }
33:
34:    cout << endl;
35:
36:    // Erase an element at the top
37:    dqIntegers.pop_front ();
38:
39:    // Erase an element at the bottom
40:    dqIntegers.pop_back ();
41:
42:    cout << "The contents of the deque after erasing an element ";
43:    cout << "from the top and bottom are:" << endl;
44:
45:    // Display contents again: this time using iterators
46:    deque <int>::iterator iElementLocator;
47:    for ( iElementLocator = dqIntegers.begin ()
48:        ; iElementLocator != dqIntegers.end ()
49:        ; ++ iElementLocator )
50:    {
51:        size_t nOffset = distance (dqIntegers.begin (), iElementLocator);
52:        cout<<"Element [" << nOffset << "] = " << *iElementLocator<<endl;
53:    }
54:
55:    return 0;
56: }
```

---

#### ▼ 输出:

---

```

The contents of the deque after inserting elements at the top and bottom are:
Element [0] = 0
Element [1] = 1
Element [2] = 2
Element [3] = 3
Element [4] = 4
Element [5] = 5
```

---

```

The contents of the deque after erasing an element from the top and bottom are:
Element [0] = 1
Element [1] = 2
Element [2] = 3
Element [3] = 4

```

### ▼ 分析:

第 10 行实例化了一个整型 deque, 其语法与实例化整型 vector 极其相似。第 13~16 行演示了 deque 的成员函数 push\_back 的用法, 然后第 18~20 行演示了 push\_front 的用法, push\_front 是 deque 唯一不同于 vector 的地方。pop\_front 的用法类似, 如第 37 行所示。要显示 deque 的内容, 第一种方法是使用数组语法访问其元素, 第二种方法是使用迭代器。使用迭代器时 (如第 47~53 行所示), 使用算法 std::distance 计算元素的偏移位置, 这与程序清单 18.6 中处理 vector 时相同。

## 18.5 总结

本章介绍了将 vector 和 deque 用作动态数组的基本知识, 还解释了大小与容量的概念。通过对 vector 进行优化, 减少了重新分配内部缓冲区的次数。重新分配缓冲区时, 需要复制容器包含的对象, 这可能降低性能。vector 是最简单的 STL 容器, 也是最常用和最高效的。

## 18.6 问与答

问: vector 会改变其存储的元素的顺序吗?

答: vector 是一种顺序容器, 元素的存储顺序与插入顺序相同。

问: 要将元素插入到 vector 中, 应使用哪个函数? 元素将插入到 vector 的什么位置?

答: 成员函数 push\_back 将元素插入到 vector 末尾。

问: 哪个函数用于获悉存储在 vector 中的元素个数?

答: 成员函数 size() 返回存储在 vector 中的元素个数。对于所有 STL 容器, 该函数都如此。

问: 随着 vector 包含的元素增多, 在 vector 末尾插入或删除元素所需的时间是否更长?

答: 否。在 vector 末尾插入或删除元素所需的时间是固定的。

问: 使用成员函数 reserve 的优点是什么?

答: reserve(...) 为 vector 的内部缓冲区分配内存空间, 这样在插入元素时 vector 就不需要重新分配缓冲区并复制现有内容。根据 vector 存储的对象类型, 为 vector 预留内存空间可能改善性能。

问: 在插入元素方面, deque 与 vector 是否不同?

答: 没有。在插入元素方面, deque 的特点与 vector 类似。将元素插入到末尾时, 两者所需的时间都是固定的, 而将元素插入到中间时, 所需的时间与容器包含的元素数成正比。然而, vector 只允许在末尾插入, 而 deque 允许在开头和末尾插入。

## 18.7 作业

作业包括测验和练习, 前者帮助读者加深对所学知识的理解, 后者提供了使用新学知识的机会。

请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 18.7.1 测验

1. 在 `vector` 的开头或中间插入元素时，所需的时间是否是固定的？
2. 有一个 `vector`，对其调用函数 `size()` 和 `capacity()` 时分别返回 10 和 20。还可再插入多少个元素而不会导致 `vector` 重新分配其缓冲区？
3. `pop_back` 函数有何功能？
4. 如果 `vector<int>` 是一个整型动态数组，那 `vector<CMammal>` 是什么类型的动态数组？
5. 能否随机访问 `vector` 中的元素？如果是，如何访问？
6. 哪种迭代器可用于随机访问 `vector` 中的元素？

### 18.7.2 练习

1. 编写一个交互式程序，它接受用户输入的整数并将其存储到 `vector` 中。用户应能够随时使用索引查询 `vector` 中存储的值。
2. 对练习 1 中的程序进行扩展，使其能够告诉用户他查询的值是否在 `vector` 中。
3. Jack 在 eBay 销售广口瓶。为帮助他打包和发货，请编写一个程序，让他能够输入每件商品的尺寸，将其存储在 `vector` 中再显示到屏幕上。



# 第 19 章

## STL list

标准模板库 (STL) 以模板类 `std::list` 的方式向程序员提供了一个双向链表。在本章中，您将学习：

- `std::list` 的特点
- 基本的 `list` 操作

### 19.1 `std::list` 的特点

链表是一系列节点，其中每个节点除包含对象或值外还指向下一个节点，即每个节点都链接到下一个节点，如图 19.1 所示。

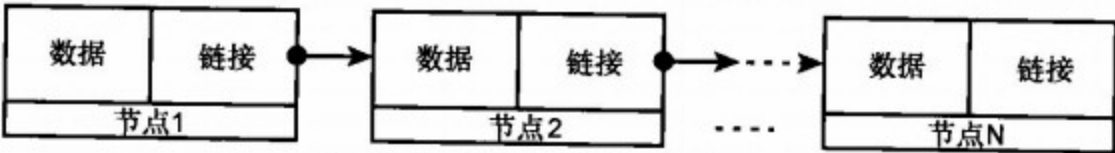


图 19.1 单向链表的可视化表示

另外，在 STL 提供的双向链表中，每个节点还指向前一个节点。因此，对于双向链表，可沿两个方向进行遍历，如图 19.2 所示。

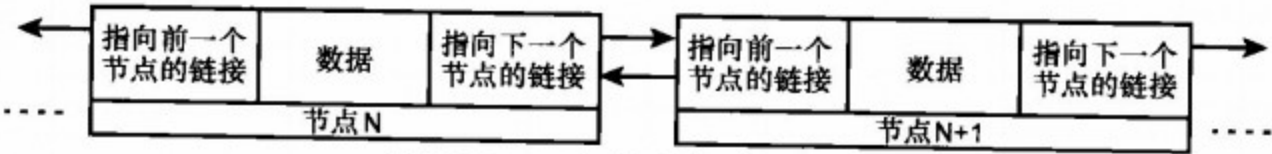


图 19.2 双向链表的可视化表示

在 STL `list` 中插入元素所需的时间都是固定的，而不管是在开头、中间还是末尾插入。

### 19.2 基本的 `list` 操作

要使用遵循标准的 STL `list` 类，必须包含头文件 `<list>`。`std` 命名空间中的模板类 `list` 是一种泛型实现，要使用其成员函数，必须实例化该模板。

#### 19.2.1 实例化 `std::list` 对象

要实例化整型 `list`，需要针对 `int` 类型具体化模板类 `std::list`，如程序清单 19.1 所示。

程序清单 19.1 实例化整型 STL list

```
1: #include <list>
2: int main ()
3: {
4:     using namespace std;
5:
6:     list <int> listIntegers;
7:     return 0;
8: }
```

#### ▼ 分析:

第 6 行针对 int 类型实例化了 STL list 类, 即定义了一个名为 listIntegers 整型 list。要实例化其他类型的 list, 可使用如下语法:

```
list <MyFavoriteType> listMyFavoriteObjects;
```

### 19.2.2 在 list 开头插入元素

要在 list 开头 (即第一个元素前面) 插入元素, 可使用 list 的成员方法 push\_front, 该方法接受一个参数——要插入的值, 如程序清单 19.2 所示。

程序清单 19.2 使用 push\_front 在 list 中插入元素

```
1: #include <list>
2: #include <iostream>
3:
4: int main ()
5: {
6:     std::list <int> listIntegers;
7:
8:     listIntegers.push_front (10);
9:     listIntegers.push_front (2001);
10:    listIntegers.push_front (-1);
11:    listIntegers.push_front (9999);
12:
13:    std::list <int> ::iterator iElementLocator;
14:
15:    for ( iElementLocator = listIntegers.begin ()
16:         ; iElementLocator != listIntegers.end ()
17:         ; ++ iElementLocator )
18:        std::cout << *iElementLocator << std::endl;
19:
20:    return 0;
21: }
```

#### ▼ 输出:

```
9999
-1
2001
10
```

#### ▼ 分析:

第 8~11 行使用 push::front 将输出显示的值 10、2 001、-1 和 9 999 依次插入到 list 开头, 因此, 第 15~18 行访问 list 的内容时, 您看到每个在开头插入的数都出现于在它之前插入的数前面, 即最新的元素放在 list 开头, 最老的元素放在 list 末尾。

### 19.2.3 在 list 末尾插入元素

要在 list 末尾 (即最后一个元素的后面) 插入元素, 可以使用 list 的成员方法 push\_back, 该方法

也只接受一个参数——要插入的值，如程序清单 19.3 所示。

程序清单 19.3 使用 push\_back 在 list 末尾插入元素

---

```

1: #include <list>
2: #include <iostream>
3:
4: int main ()
5: {
6:     std::list<int> listIntegers;
7:
8:     listIntegers.push_back (10);
9:     listIntegers.push_back (2001);
10:    listIntegers.push_back (-1);
11:    listIntegers.push_back (9999);
12:
13:    std::list<int> ::iterator iElementLocator;
14:
15:    for ( iElementLocator = listIntegers.begin ()
16:          ; iElementLocator != listIntegers.end ()
17:          ; ++ iElementLocator )
18:        std::cout << *iElementLocator << std::endl;
19:
20:    return 0;
21: }
```

---

#### ▼ 输出:

```

10
2001
-1
9999
```

#### ▼ 分析:

第 8~11 行使用 push\_back 在 list 中插入元素，并将其放在之前插入的元素后面。在屏幕上显示 list 的内容时（如第 15~18 行所示），这些元素的排列顺序与使用 push\_back 在 list 末尾插入的顺序相同。

## 19.2.4 在 list 中间插入元素

std::list 的特点之一是，在其中间插入元素所需的时间是固定的，这项工作是由成员函数 insert 完成的。

成员函数 list::insert 有 3 种版本。

第 1 种版本:

```
iterator insert(iterator pos, const T& x)
```

在这里，insert 函数接受的第 1 个参数是插入位置，第 2 个参数是要插入的值。该函数返回一个迭代器，它指向刚插入到 list 中的元素。

第 2 种版本:

```
void insert(iterator pos, size_type n, const T& x)
```

该函数的第 1 个参数是插入位置，最后一个参数是要插入的值，而第 2 个参数是要插入的元素个数。

第 3 种版本:

```
template <class InputIterator>
void insert(iterator pos, InputIterator f, InputIterator l)
```

该重载版本是一个模板函数，它除接受一个位置参数外，还接受两个输入迭代器，指定要将集合中相应范围内的元素插入到 list 中。注意，输入类型 InputIterator 是一种模板参数化类型，因此可指定任何集合——数组、vector 或另一个 list 的边界。

程序清单 19.4 演示了如何使用函数 list::insert 的这些重载版本。

程序清单 19.4 在 list 中插入元素的各种方法

```

1: #include <list>
2: #include <iostream>
3:
4: using namespace std;
5:
6: void PrintListContents (const list <int>& listInput);
7:
8: int main ()
9: {
10:     list <int> listIntegers1;
11:
12:     // Inserting elements at the beginning...
13:     listIntegers1.insert (listIntegers1.begin (), 4);
14:     listIntegers1.insert (listIntegers1.begin (), 3);
15:     listIntegers1.insert (listIntegers1.begin (), 2);
16:     listIntegers1.insert (listIntegers1.begin (), 1);
17:
18:     // Inserting an element at the end...
19:     listIntegers1.insert (listIntegers1.end (), 5);
20:
21:     cout << "The contents of list 1 after inserting elements:" << endl;
22:     PrintListContents (listIntegers1);
23:
24:     list <int> listIntegers2;
25:
26:     // Inserting 4 elements of the same value 0...
27:     listIntegers2.insert (listIntegers2.begin (), 4, 0);
28:
29:     cout << "The contents of list 2 after inserting ";
30:     cout << listIntegers2.size () << " elements of a value:" << endl;
31:     PrintListContents (listIntegers2);
32:
33:     list <int> listIntegers3;
34:
35:     // Inserting elements from another list at the beginning...
36:     listIntegers3.insert (listIntegers3.begin (),
37:                          listIntegers1.begin (), listIntegers1.end ());
38:
39:     cout << "The contents of list 3 after inserting the contents of ";
40:     cout << "list 1 at the beginning:" << endl;
41:     PrintListContents (listIntegers3);
42:
43:     // Inserting elements from another list at the end...
44:     listIntegers3.insert (listIntegers3.end (),
45:                          listIntegers2.begin (), listIntegers2.end ());
46:
47:     cout << "The contents of list 3 after inserting ";
48:     cout << "the contents of list 2 at the beginning:" << endl;
49:     PrintListContents (listIntegers3);
50:
51:     return 0;
52: }
53:
54: void PrintListContents (const list <int>& listInput)
55: {
56:     // Write values to the screen...
57:     cout << "{ ";
58:
59:     std::list <int>::const_iterator iElementLocator;
60:     for ( iElementLocator = listInput.begin ()
61:          ; iElementLocator != listInput.end ()
62:          ; ++ iElementLocator )
63:         cout << *iElementLocator << " ";
64:
65:     cout << "}" << endl << endl;
66: }

```

## ▼ 输出:

The contents of list 1 after inserting elements:

```
{ 1 2 3 4 5 }
```

```
The contents of list 2 after inserting '4' elements of a value:
{ 0 0 0 0 }
```

```
The contents of list 3 after inserting the contents of list 1 at the beginning:
{ 1 2 3 4 5 }
```

```
The contents of list 3 after inserting the contents of list 2 at the beginning:
{ 1 2 3 4 5 0 0 0 0 }
```

### ▼ 分析:

在程序清单 19.4 中, `begin()` 和 `end()` 是 `list` 的成员函数, 它们分别返回指向 `list` 开头和末尾的迭代器。`list::insert` 函数接受一个迭代器参数, 元素将插入到该参数指定的位置前面。`end()` 函数返回的迭代器 (如第 19 行所示) 指向 `list` 中最后一个元素的后面, 因此该行将值 5 插入到末尾。

第 36 行和第 44 行演示了如何将一个 `list` 的内容插入到另一个 `list` 中。注意, 这里使用的 `insert` 函数接受的参数类型为 `InputIterator`。虽然这里演示的是将一个整型 `list` 插入到另一个 `list` 中, 但也可使用 `vector` 或普通静态数组的边界指定插入范围。

## 19.2.5 删除 `list` 中的元素

`list` 的成员函数 `erase` 有两种重载版本: 一个接受一个迭代器参数并删除迭代器指向的元素, 另一个接受两个迭代器参数并删除指定范围内的所有元素。程序清单 19.5 演示了如何使用 `list::erase` 函数删除一个元素或指定范围内的所有元素。

程序清单 19.5 删除 `list` 中的元素

```
1: #include <list>
2: #include <iostream>
3:
4: using namespace std;
5:
6: void PrintListContents (const list <int>& listInput);
7:
8: int main ()
9: {
10:     std::list <int> listIntegers;
11:
12:     // Insert elements at the beginning...
13:     listIntegers.push_front (4);
14:     listIntegers.push_front (3);
15:
16:     // Store an iterator obtained in using the 'insert' function
17:     list <int>::iterator iElementValueTwo;
18:     iElementValueTwo = listIntegers.insert (listIntegers.begin (), 2);
19:
20:     listIntegers.push_front (1);
21:     listIntegers.push_front (0);
22:
23:     // Insert an element at the end...
24:     listIntegers.push_back (5);
25:
26:     cout << "Initial contents of the list:" << endl;
27:     PrintListContents (listIntegers);
28:
29:     listIntegers.erase (listIntegers.begin (), iElementValueTwo);
30:     cout << "Contents after erasing a range of elements:" << endl;
31:     PrintListContents (listIntegers);
32:
33:     cout << "Contents after erasing element
34:     '" << *iElementValueTwo << "':" << endl;
35:     listIntegers.erase (iElementValueTwo);
36:     PrintListContents (listIntegers);
37: }
```



```

37:     listIntegers.erase (listIntegers.begin (), listIntegers.end ());
38:     cout << "Contents after erasing a range:" << endl;
39:     PrintListContents (listIntegers);
40:
41:     return 0;
42: }
43:
44: void PrintListContents (const list <int>& listInput)
45: {
46:     if (listInput.size () > 0)
47:     {
48:         // Write values to the screen...
49:         cout << "{ ";
50:
51:         std::list <int>::const_iterator iElementLocator;
52:         for ( iElementLocator = listInput.begin ()
53:               ; iElementLocator != listInput.end ()
54:               ; ++ iElementLocator )
55:             cout << *iElementLocator << " ";
56:
57:         cout << "}" << endl << endl;
58:     }
59:     else
60:         cout << "List is empty!" << endl;
61: }

```

#### ▼ 输出:

```

Initial contents of the list:
{ 0 1 2 3 4 5 }

Contents after erasing a range of elements:
{ 2 3 4 5 }

Contents after erasing element '2':
{ 3 4 5 }

Contents after erasing a range:
List is empty!

```

#### ▼ 分析:

第 18 行使用 list 的成员函数 insert (而不是像其他地方那样使用 push\_front) 将一个元素插入到 list 开头, 这是因为 insert 函数返回一个迭代器, 它指向刚插入的元素 (该迭代器被存储到 iElementValueTwo 中)。如第 29 行所示, 该迭代器随后被用于定义要从 list 中删除的元素范围: 从 list 开头到 iElementValueTwo 指向的元素, 但不包括 iElementValueTwo。第 34 行演示了 list 的成员函数 erase 的另一版本的使用法, 该版本删除迭代器 iElementValueTwo 指向的元素。在使用 iElementValueTwo 调用 erase 函数前, 第 33 行显示了它指向的值。这里是有意这样做的, 因为调用 erase 后, 该迭代器将无效。

第 37 行使用的 erase 成员函数将整个 list 清空, 因为使用 begin() 和 end() 指定的范围包含 list 的所有元素。然而, list 还提供了 clear() 函数, 它与这种调用 erase 函数的方式等效: 清空整个集合。

## 19.3 对 list 中元素进行反转和排序

list 的一个独特之处是, 指向元素的迭代器在 list 的元素重新排列或插入元素后仍有效。为实现这种特点, list 提供了成员方法 sort 和 reverse, 虽然标准模板库也提供了这两种算法, 它们也可用于 list 类。这些算法的成员函数版本确保元素的相对位置发生变化后指向元素的迭代器仍有效。

### 19.3.1 反转元素的排列顺序

list 提供了成员函数 reverse(), 该函数没有参数, 它反转 list 中元素的排列顺序, 如程序清单 19.6 所示。

程序清单 19.6 反转 list 中元素的排列顺序

```

1: #include <list>
2: #include <iostream>
3:
4: using namespace std;
5:
6: void PrintListContents (const list <int>& listInput);
7:
8: int main ()
9: {
10:     std::list <int> listIntegers;
11:
12:     // Insert elements at the beginning...
13:     listIntegers.push_front (4);
14:     listIntegers.push_front (3);
15:     listIntegers.push_front (2);
16:
17:     listIntegers.push_front (1);
18:     listIntegers.push_front (0);
19:
20:     // Insert an element at the end...
21:     listIntegers.push_back (5);
22:
23:     cout << "Initial contents of the list:" << endl;
24:     PrintListContents (listIntegers);
25:
26:     listIntegers.reverse ();
27:
28:     cout << "Contents of the list after using reverse ():" << endl;
29:     PrintListContents (listIntegers);
30:
31:     return 0;
32: }
33:
34: void PrintListContents (const list <int>& listInput)
35: {
36:     if (listInput.size () > 0)
37:     {
38:         // Write values to the screen...
39:         cout << "{ ";
40:
41:         std::list <int>::const_iterator iElementLocator;
42:         for ( iElementLocator = listInput.begin ()
43:              ; iElementLocator != listInput.end ()
44:              ; ++ iElementLocator )
45:             cout << *iElementLocator << " ";
46:
47:         cout << "}" << endl << endl;
48:     }
49:     else
50:         cout << "List is empty!" << endl;
51: }

```

**▼ 输出:**

```
Initial contents of the list:
{ 0 1 2 3 4 5 }
```

```
Contents of the list after using reverse ():
{ 5 4 3 2 1 0 }
```

**▼ 分析:**

如第 26 行所示, `reverse()` 只是反转 list 中元素的排列顺序。它是一个没有参数的简单函数, 确保指向元素的迭代器在反转后仍有效——如果程序员保存了该迭代器。

### 19.3.2 元素排序

list 的成员函数 `sort()` 有两个版本: 一个没有参数, 另一个接受一个二元谓词函数作为参数, 将根

据该谓词指定的标准进行排序。

使用前一个版本时，程序员必须为 list 包含的元素所属的类实现运算符<，让 list 的 sort 函数根据其要求进行排序。如果没有提供排序谓词，std::list 将通过 std::less 调用运算符<对元素进行比较，以便将元素重新排序。程序员也可向 list::sort 提供第二个参数——一个二元谓词。二元谓词是一个函数，它接受两个输入值，并返回一个布尔值指出第一个值是否比第二个值小。程序清单 19.7 演示了这些技巧。

程序清单 19.7 将整型 list 按升序和降序排列

```
1: #include <list>
2: #include <iostream>
3:
4: using namespace std;
5:
6: void PrintListContents (const list <int>& listInput);
7: bool SortPredicate_Descending (const int& lsh, const int& rsh);
8:
9: int main ()
10: {
11:     std::list <int> listIntegers;
12:
13:     // Insert elements at the beginning...
14:     listIntegers.push_front (444);
15:     listIntegers.push_front (300);
16:     listIntegers.push_front (21111);
17:
18:     listIntegers.push_front (-1);
19:     listIntegers.push_front (0);
20:
21:     // Insert an element at the end...
22:     listIntegers.push_back (-5);
23:
24:     cout << "Initial contents of the list are - " << endl;
25:     PrintListContents (listIntegers);
26:
27:     listIntegers.sort ();
28:
29:     cout << "Order of elements after sort(): " << endl;
30:     PrintListContents (listIntegers);
31:
32:     listIntegers.sort (SortPredicate_Descending);
33:     cout << "Order of elements after sort() with a predicate: " << endl;
34:
35:     PrintListContents (listIntegers);
36:
37:     return 0;
38: }
39:
40: void PrintListContents (const list <int>& listInput)
41: {
42:     if (listInput.size () > 0)
43:     {
44:         // Write the output...
45:         cout << "{ ";
46:
47:         std::list <int>::const_iterator iElementLocator;
48:         for ( iElementLocator = listInput.begin ()
49:              ; iElementLocator != listInput.end ()
50:              ; ++ iElementLocator )
51:             cout << *iElementLocator << " ";
52:
53:         cout << "}" << endl << endl;
54:     }
55:     else
56:         cout << "List is empty!" << endl;
57: }
58:
59: bool SortPredicate_Descending (const int& lsh, const int& rsh)
60: {
61:     return (rsh < lsh);
62: }
```

## ▼ 输出:

```
Initial contents of the list are -
{ 0 -1 21111 300 444 -5 }

Order of elements after sort():
{ -5 -1 0 300 444 21111 }

Order of elements after sort() with a predicate:
{ 21111 444 300 0 -1 -5 }
```

## ▼ 分析:

该示例演示了如何对整型 list 进行排序。开头几行代码创建了一个 list 对象,并在其中插入一些值。第 27 行演示了不带参数的 sort() 函数的用法,它使用运算符<比较整数,并将元素按默认的升序排列。然而,如果程序员要覆盖这种默认行为,它必须向 sort 函数提供一个二元谓词。第 59~62 行定义了函数 SortPredicate\_Descending,它是一个二元谓词,帮助 list 的 sort 函数判断一个元素是否比另一个元素小。如果不是,则交换这两个元素的位置。将这个函数作为参数传递给了 sort() 函数,如第 32 行所示。从本质上说,这个二元谓词根据应用程序的需求定义了两个元素之间的小于关系。由于这里的需求是按降序排列,因此这个谓词仅在第一个值比第二个值大时返回 true。也就是说,根据这里的逻辑,仅当第一个元素 (lsh) 的数字值比第二个元素 (rsh) 大时,才认为第一个元素比第二个元素小。

## 注意

谓词将在第 22 章详细讨论, STL std::sort 算法以及众多其他的算法将在第 23 章讨论。

在实际的应用程序中,很少使用 STL 容器来存储整数,而使用它们来存储用户定义的类型,如类或结构。程序清单 19.8 就是一个这样的应用程序,其中 list 包含联系人信息。

程序清单 19.8 存储结构的 list: 创建一个联系人列表

```
1: #include <list>
2: #include <string>
3: #include <iostream>
4:
5: using namespace std;
6:
7: enum MenuOptionSelection
8: {
9:     InsertContactListEntry = 0,
10:    SortOnName = 1,
11:    SortOnNumber = 2,
12:    DisplayEntries = 3,
13:    EraseEntry = 4,
14:    QuitContactList = 5
15: };
16:
17: struct ContactListItem
18: {
19:     string strContactsName;
20:     string strPhoneNumber;
21:
22:     // Constructor and destructor
23:     ContactListItem (const string& strName, const string & strNumber)
24:     {
25:         strContactsName = strName;
26:         strPhoneNumber = strNumber;
27:     }
28:
29:     bool operator == (const ContactListItem& itemToCompare) const
30:     {
31:         return (itemToCompare.strContactsName == this->strContactsName);
32:     }
33:
34:     bool operator < (const ContactListItem& itemToCompare) const
35:     {
36:         return (this->strContactsName < itemToCompare.strContactsName);
37:     }
38: };
```

```
39:
40:
41: int ShowMenu ();
42: ContactListItem GetContactInfo ();
43: void DisplayContactList (const list <ContactListItem>& listContacts);
44: void EraseEntryFromList (list <ContactListItem>& listContacts);
45: bool Predicate_CheckItemsOnNumber (const ContactListItem& item1,
46:                                     const ContactListItem& item2);
47:
48: int main ()
49: {
50:     list <ContactListItem> listContacts;
51:     int nUserSelection = 0;
52:
53:     while ((nUserSelection = ShowMenu ()) != (int) QuitContactList)
54:     {
55:         switch (nUserSelection)
56:         {
57:             case InsertContactListEntry:
58:                 listContacts.push_back (GetContactInfo ());
59:                 cout << "Contacts list updated!" << endl << endl;
60:                 break;
61:
62:             case SortOnName:
63:                 listContacts.sort ();
64:                 DisplayContactList (listContacts);
65:                 break;
66:
67:             case SortOnNumber:
68:                 listContacts.sort (Predicate_CheckItemsOnNumber);
69:                 DisplayContactList (listContacts);
70:                 break;
71:
72:             case DisplayEntries:
73:                 DisplayContactList (listContacts);
74:                 break;
75:
76:             case EraseEntry:
77:                 EraseEntryFromList (listContacts);
78:                 DisplayContactList (listContacts);
79:                 break;
80:
81:             case QuitContactList:
82:                 cout << "Ending application, bye!" << endl;
83:                 break;
84:
85:             default:
86:                 cout << "Invalid input '" << nUserSelection << ".'";
87:                 cout << "Choose an option between 0 and 4" << endl << endl;
88:                 break;
89:         }
90:     }
91:
92:     cout << "Quitting! Bye!" << endl;
93:
94:     return 0;
95: }
96:
97: int ShowMenu ()
98: {
99:     cout << "**** What would you like to do next? ****" << endl << endl;
100:    cout << "Enter 0 to feed a name and phone number" << endl;
101:    cout << "Enter 1 to sort the list by name" << endl;
102:    cout << "Enter 2 to sort the list by number" << endl;
103:    cout << "Enter 3 to Display all entries" << endl;
104:    cout << "Enter 4 to erase an entry" << endl;
105:    cout << "Enter 5 to quit this application" << endl << endl;
106:    cout << "> ";
107:
108:    int nOptionSelected = 0;
109:
110:    // Accept user input
111:    cin >> nOptionSelected ;
```



```

112:
113:     cout << endl;
114:     return nOptionSelected;
115: }
116:
117: bool Predicate_CheckItemsOnNumber (const ContactListItem& item1,
118:                                     const ContactListItem& item2)
119: {
120:     return (item1.strPhoneNumber < item2.strPhoneNumber);
121: }
122:
123: ContactListItem GetContactInfo ()
124: {
125:     cout << "*** Feed contact information ***" << endl;
126:     string strName;
127:     cout << "Please enter the person's name" << endl;;
128:     cout << "> ";
129:     cin >> strName;
130:
131:     string strPhoneNumber;
132:     cout << "Please enter "<< strName << "'s phone number" << endl;
133:     cout << "> ";
134:     cin >> strPhoneNumber;
135:
136:     return ContactListItem (strName, strPhoneNumber);
137: }
138:
139: void DisplayContactList (const list <ContactListItem>& listContacts)
140: {
141:     cout << "*** Displaying contact information ***" << endl;
142:     cout << "There are " << listContacts.size ();
143:     cout << " entries in the contact-list" << endl;
144:
145:     list <ContactListItem>::const_iterator iContact;
146:     for ( iContact = listContacts.begin ()
147:           ; iContact != listContacts.end ()
148:           ; ++ iContact )
149:     {
150:         cout << "Name: '" << iContact->strContactsName;
151:         cout << "' Number: '" << iContact->strPhoneNumber << "'" << endl;
152:     }
153:
154:     cout << endl;
155: }
156:
157: void EraseEntryFromList (list <ContactListItem>& listContacts)
158: {
159:     cout << "*** Erase an entry ***" << endl;
160:     cout << "Enter the name of the contact you wish to delete" << endl;
161:     cout << "> ";
162:     string strNameToErase;
163:     cin >> strNameToErase;
164:
165:     listContacts.remove (ContactListItem (strNameToErase, ""));
166: }

```

### ▼ 输出:

```
*** What would you like to do next? ***
```

```

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

```

```
> 0
```

```

*** Feed contact information ***
Please enter the person's name
> John
Please enter John's phone number

```

```
> 989812132
Contacts list updated!

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 0

*** Feed contact information ***
Please enter the person's name
> Alanis
Please enter Alanis's phone number
> 78451245
Contacts list updated!

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 0

*** Feed contact information ***
Please enter the person's name
> Tim
Please enter Tim's phone number
> 45121655
Contacts list updated!

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 0

*** Feed contact information ***
Please enter the person's name
> Ronald
Please enter Ronald's phone number
> 123456987
Contacts list updated!

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 3

*** Displaying contact information ***
There are 4 entries in the contact-list
Name: 'John' Number: '989812132'
Name: 'Alanis' Number: '78451245'
Name: 'Tim' Number: '45121655'
```

```

Name: 'Ronald' Number: '123456987'

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 1

*** Displaying contact information ***
There are 4 entries in the contact-list
Name: 'Alanis' Number: '78451245'
Name: 'John' Number: '989812132'
Name: 'Ronald' Number: '123456987'
Name: 'Tim' Number: '45121655'

*** What would you like to do next? ***
Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 2

*** Displaying contact information ***
There are 4 entries in the contact-list
Name: 'Ronald' Number: '123456987'
Name: 'Tim' Number: '45121655'
Name: 'Alanis' Number: '78451245'
Name: 'John' Number: '989812132'

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 4

*** Erase an entry ***
Enter the name of the contact you wish to delete
> Tim
*** Displaying contact information ***
There are 3 entries in the contact-list
Name: 'Ronald' Number: '123456987'
Name: 'Alanis' Number: '78451245'
Name: 'John' Number: '989812132'

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to sort the list by name
Enter 2 to sort the list by number
Enter 3 to Display all entries
Enter 4 to erase an entry
Enter 5 to quit this application

> 5

Quitting! Bye!

```

### ▼ 分析:

这个示例用到本章介绍的全部知识，并将它们应用于一个包含 `ContactListItem` 对象的 `list`。第 17~38

行定义了结构 `ContactListItem`，该结构包含两个成员变量：一个用于存储人名，另一个用于存储电话号码。它还实现了运算符 `=` 和 `<`，前者帮助 STL 算法和 `list` 的成员函数（如 `remove`）将元素从 `list` 中删除（如第 165 行所示），后者帮助 STL 算法和 `list` 的成员函数（如 `sort`）对 `list` 中的 `ContactListItem` 对象进行排序（如第 63 行所示）。运算符 `<` 为 `list` 中的对象提供一种默认排序标准（即根据人名的词典顺序），但如果 `list` 需要根据其他标准（如电话号码）排序，可使用接受二元谓词的 `sort` 函数，如第 68 行所示。这个二元谓词是一个简单的函数，它接受两个值，并返回一个布尔值指出第一个值是否小于第二个值。`sort` 函数使用这个二元谓词比较两个值，并根据谓词的返回值决定是否交换两个元素的位置（从而将它们排序）。在程序清单 19.8 中，第 35 行声明了函数 `Predicate_CheckItemsOnNumber`，第 92~95 行对其进行定义，该函数用于根据结构 `ContactListItem` 的 `strPhoneNumber` 属性对 `list` 存储的对象进行排序。

这个例子表明 STL `list` 是一个模板类，可用于创建任何对象类型的列表，它还说明了运算符与谓词的重要性。

## 19.4 总结

本章介绍 `list` 的特征以及各种 `list` 操作。现在读者知道了 `list` 的最常用函数，能够创建用于存储任何类型的对象的 `list`。

## 19.5 问与答

问：`list` 为何提供诸如 `sort` 和 `remove` 等成员函数？

答：STL `list` 类需要确保指向 `list` 中元素的迭代器始终有效，而不管如何在 `list` 中移动该元素。虽然 STL 算法也可用于 `list`，但 `list` 的成员函数可确保 `list` 的上述特征，即将 `list` 排序后，指向 `list` 中元素的迭代器仍指向原来的元素。

问：使用存储 `CAnimal` 对象的 `list` 时，为让 `list` 的成员函数能够正确处理 `CAnimal` 对象，应为 `CAnimal` 类实现哪些运算符？

答：对于其对象将存储在 STL 容器中的类，必须为它实现默认比较运算符 `=` 和运算符 `<`。为覆盖其默认行为，可给 `std::list` 的成员函数 `sort` 提供一个排序谓词。

## 19.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 19.6.1 测验

1. 与在开头或末尾插入元素相比，在 STL `list` 中间插入元素是否会降低性能？
2. 假设有两个迭代器分别指向 STL `list` 对象中的两个元素，然后在这两个元素之间插入了一个元素。请问这种插入是否会导致这两个迭代器无效？
3. 如何清空 `std::list` 的内容？
4. 能否在 `list` 中插入多个元素？

### 19.6.2 练习

1. 编写一个程序，它接受用户输入的数字并将它们插入到 `list` 开头。
2. 使用一个简短的程序来演示这样一点：在 `list` 中插入一个新元素，导致迭代器指向的元素的相对位置发生变化后，该迭代器仍有效。
3. 编写一个程序，使用 `list` 的 `insert` 函数将一个 `vector` 的内容插入到一个 STL `list` 中。
4. 编写一个程序，对字符串 `list` 进行排序以及反转排列顺序。

## 第 20 章

# STL set 与 multiset

标准模板库 (STL) 向程序员提供了一些容器类, 以便在应用程序中进行频繁而快速的搜索。在本章中, 您将学习:

- STL set 和 multiset 简介
- STL set 和 multiset 的基本操作
- 使用 STL set 和 multiset 容器的优缺点

### 20.1 简介

容器 set 和 multiset 让程序员能够快速查找键, 键是存储在一维容器中的值。set 和 multiset 之间的区别在于, 后者可存储重复的值, 而前者只能存储唯一的值。

图 20.1 表明, set 只能包含唯一的人名, 而 multiset 可存储重复的人名。STL 容器是泛型模板类, 因此可包含字符串, 也可包含整型、结构或类对象, 这取决于如何实例化模板。

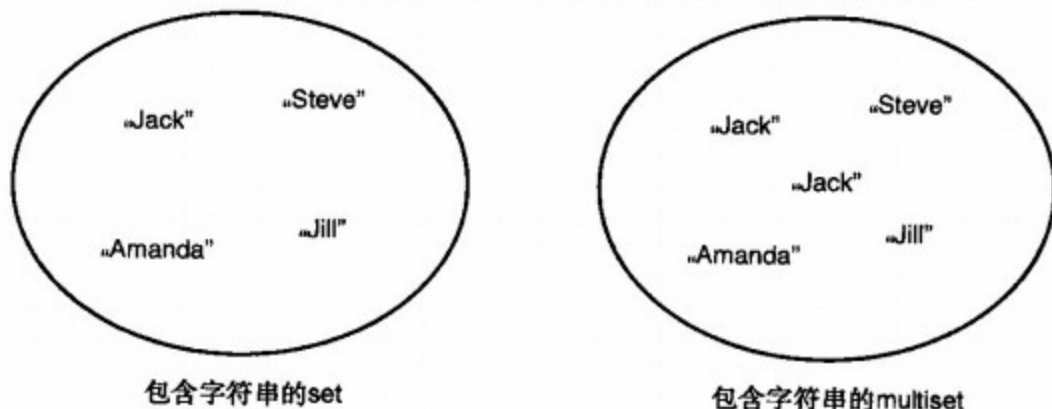


图 20.1 包含人名的 set 和 multiset 的可视化表示

为实现快速搜索, STL set 和 multiset 的内部结构像一棵二叉树, 这意味着将元素插入到 set 或 multiset 时将其进行排序, 以提高查找速度。这还意味着不像 vector 那样可以使用其他元素替换给定位置的元素, 位于 set 中特定位置的元素不能替换为值不同的新元素, 这是因为 set 将把新元素同二叉树中的其他元素进行比较, 进而将其放在其他位置。

### 20.2 STL set 和 multiset 的基本操作

STL set 和 multiset 都是模板类, 要使用其成员函数, 必须先实例化。要使用 STL set 或 multiset 类, 程序必须包含头文件 <set>。

#### 20.2.1 实例化 std::set 对象

要实例化一个整型 set 或 multiset, 必须针对 int 类型具体化模板类 std::set 或 std::multiset, 如程序清单 20.1 所示。



程序清单 20.1 实例化整型 STL set 和 multiset

```

1: #include <set>
2:
3: int main ()
4: {
5:     using namespace std;
6:
7:     set <int> setIntegers;
8:     multiset <int> msetIntegers;
9:
10:    return 0;
11: }

```

## ▼ 分析:

第 1 行指出了要使用 STL 类 set 和 multiset 必须包含的标准头文件, 该文件位于 std 命名空间内。第 7 行和第 8 行演示了如何针对 int 类型实例化这两个模板类。

## 提示

以这种形式创建的 set 将使用 `std::less <T>` 进行排序。这是默认排序谓词, 它调用运算符 `<`。在有些应用程序中, 可能需要覆盖排序机制, 为此可在实例化 set 时, 将可选的第二个参数指定为排序谓词 (二元谓词), 如下所示:

```

set <ObjectType, OptionalBinaryPredicate>
setWithCustomSortPredicate;

```

## 20.2.2 在 STL set 或 multiset 中插入元素

set 和 multiset 的大多数函数的用法类似, 它们接受类似的参数, 返回类型也类似, 例如, 要在这两种容器中插入元素, 都可使用成员函数 insert, 这个函数接受要插入的值。insert 函数还有其他版本, 程序清单 20.2 演示了其中的一个。

程序清单 20.2 在 STL set 或 multiset 中插入元素

```

1: #include <set>
2: #include <iostream>
3: using namespace std;
4:
5: template <typename Container>
6: void PrintContents (const Container & stlContainer);
7:
8: int main ()
9: {
10:    set <int> setIntegers;
11:    multiset <int> msetIntegers;
12:
13:    setIntegers.insert (60);
14:    setIntegers.insert (-1);
15:    setIntegers.insert (3000);
16:    cout << "Writing the contents of the set to the screen" << endl;
17:    PrintContents (setIntegers);
18:
19:    msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
20:    msetIntegers.insert (3000);
21:
22:    cout << "Writing the contents of the multiset to the screen" << endl;
23:    PrintContents (msetIntegers);
24:
25:    cout << "Number of instances of '3000' in the multiset are: ";
26:    cout << msetIntegers.count (3000) << " " << endl;
27:
28:    return 0;
29: }
30:
31: template <typename Container>
32: void PrintContents (const Container & stlContainer)
33: {
34:    Container::const_iterator iElementLocator = stlContainer.begin ();
35:

```

```

36:     while (iElementLocator != stlContainer.end ())
37:     {
38:         cout << *iElementLocator << endl;
39:         ++ iElementLocator;
40:     }
41:
42:     cout << endl;
43: }

```

#### ▼ 输出:

```

Writing the contents of the set to the screen
-1
60
3000
Writing the contents of the multiset to the screen
-1
60
3000
3000

Number of instances of '3000' in the multiset are: '2'

```

#### ▼ 分析:

第5行和第6行包含模板函数 PrintContents 的声明, 该函数是在第31~43行定义的, 它将 STL 容器的内容显示到控制台 (屏幕)。第10行和第11行定义了一个 set 对象和一个 multiset 对象。第13~15行调用成员函数 insert 将值插入到 set 中。第19行演示了如何使用 insert 函数将一个容器的内容插入到另一个容器中, 这里是将 setIntegers 的内容插入到 multiset msetIntegers 中。将 set 的内容插入到 multiset 中后 (第19行), 插入一个值为 3 000 的元素, 这个元素已存储于 multiset 中, 它是在第15行插入的。从输出可知, multiset 能够存储多个相同的值。第25行和第26行演示了成员函数 multiset::count() 的用法, 它返回 multiset 中有多少个元素存储了特定的值。

### 20.2.3 在 STL set 或 multiset 中查找元素

诸如 set、multiset、map 和 multimap 等关联容器都提供了成员函数 find(), 它让您能够根据给定的键来查找值, 如程序清单 20.3 所示。对于 multiset, 这个函数将查找第一个与给定键匹配的值。

程序清单 20.3 使用成员函数 find

```

1: #include <set>
2: #include <iostream>
3:
4: using namespace std;
5: typedef set <int> SETINT;
6:
7: int main ()
8: {
9:     SETINT setIntegers;
10:
11:     // Insert some random values
12:     setIntegers.insert (43);
13:     setIntegers.insert (78);
14:     setIntegers.insert (-1);
15:     setIntegers.insert (124);
16:
17:     SETINT::const_iterator iElement;
18:
19:     // Write contents of the set to the screen
20:     for ( iElement = setIntegers.begin ()
21:         ; iElement != setIntegers.end ()
22:         ; ++ iElement )
23:         cout << *iElement << endl;
24:
25:     // Try finding an element
26:     SETINT::iterator iElementFound = setIntegers.find (-1);
27:
28:     // Check if found...
29:     if (iElementFound != setIntegers.end ())
30:         cout << "Element " << *iElementFound << " found!" << endl;

```

```

31:     else
32:         cout << "Element not found in set!" << endl;
33:
34:     // Try finding another element
35:     SETINT::iterator iAnotherFind = setIntegers.find (12345);
36:
37:     // Check if found...
38:     if (iAnotherFind != setIntegers.end ())
39:         cout << "Element " << *iAnotherFind << " found!" << endl;
40:     else
41:         cout << "Element 12345 not found in set!" << endl;
42:
43:     return 0;
44: }

```

#### ▼ 输出:

```

-1
43
78
124
Element -1 found!
Element 12345 not found in set!

```

#### ▼ 分析:

第 5 行给整型 set 指定了一个别名, 以简化使用 set 的语法, 这还可能提高代码的可读性。第 26~32 行演示了成员函数 find 的用法。第 29 行将 find() 返回的迭代器与 end() 进行比较, 以核实是否找到了指定的元素。

#### 注意

程序清单 20.3 所示的示例也适用于 multiset, 只需在第 5 行给 multiset 而不是 set 指定一个别名即可, 这不会影响应用程序:

```
typedef multiset <int> MSETINT;
```

## 20.2.4 删除 STL set 或 multiset 中的元素

诸如 set、multiset、map 和 multimap 等关联容器都提供了成员函数 erase(), 它让您能够根据键删除值:

```
setObject.erase (key);
```

erase 函数的另一个版本接受一个迭代器作为参数, 并删除迭代器指向的元素:

```
setObject.erase (iElement);
```

通过使用迭代器指定的边界, 可将指定范围内的所有元素都从 set 或 multiset 中删除:

```
setObject.erase (iLowerBound, iUpperBound);
```

程序清单 20.4 演示了如何使用 erase() 来删除 mset 或 multiset 中的元素。

#### 程序清单 20.4 使用 multiset 的成员函数 erase

```

1: #include <set>
2: #include <iostream>
3:
4: using namespace std;
5: typedef multiset <int> MSETINT;
6:
7: int main ()
8: {
9:     MSETINT msetIntegers;
10:
11:     // Insert some random values
12:     msetIntegers.insert (43);
13:     msetIntegers.insert (78);
14:     msetIntegers.insert (78);    // Duplicate
15:     msetIntegers.insert (-1);
16:     msetIntegers.insert (124);
17:
18:     MSETINT::const_iterator iElement;
19:
20:     cout << "multiset contains " << msetIntegers.size () << " elements.";
21:     cout << " These are: " << endl;
22:
23:     // Write contents of the multiset to the screen
24:     for ( iElement = msetIntegers.begin ()

```

```

25:         ; iElement != msetIntegers.end ()
26:         ; ++ iElement )
27:         cout << *iElement << endl;
28:
29:     cout << "Please enter a number to be erased from the set" << endl;
30:     int nNumberToErase = 0;
31:     cin >> nNumberToErase;
32:
33:     cout << "Erasing " << msetIntegers.count (nNumberToErase);
34:     cout << " instances of value " << nNumberToErase << endl;
35:
36:     // Try finding an element
37:     msetIntegers.erase (nNumberToErase);
38:
39:     cout << "multiset contains " << msetIntegers.size () << " elements.";
40:     cout << " These are: " << endl;
41:     for ( iElement = msetIntegers.begin ()
42:         ; iElement != msetIntegers.end ()
43:         ; ++ iElement )
44:         cout << *iElement << endl;
45:
46:     return 0;
47: }

```

### ▼ 输出:

```

multiset contains 5 elements. These are:
-1
43
78
78
124
Please enter a number to be erased from the set
78
Erasing 2 instances of value 78
multiset contains 3 elements. These are:
-1
43
124

```

### ▼ 分析:

这个例子通过调用 multiset 而不是在 set (虽然语法没有什么不同) 的成员 erase 函数, 演示了 erase() 对容器中值相同的多个元素的影响——这只有在 multiset 中才能做到。第 12~16 行在 multiset 中插入值, 其中有一个值是重复的。程序让用户输入要从 multiset 中删除的值, 如第 29~31 行所示。从输出可知, 用户输入的值是 78, 这个值在容器中出现了多次。第 37 行进行 erase 操作, 然后显示 multiset 的内容, 输出表明 multiset 删除了值为 78 的所有元素。

### 注意

函数 erase() 被重载了。可对迭代器 (如 find 返回的迭代器) 调用 erase() 以删除找到的元素, 如下所示:

```

MSETINT::iterator iElementFound = msetIntegers.find (nNumberToErase);
if (iElementFound != msetIntegers.end ())
    msetIntegers.erase (iElementFound);
else
    cout << "Element not found!" << endl;

```

同样, erase() 还可用于从 multiset 中删除指定范围内的元素:

```

MSETINT::iterator iElementFound = msetIntegers.find (nValue);

if (iElementFound != msetIntegers.end ())
    msetIntegers.erase (msetIntegers.begin (), iElementFound);

```

上述代码删除从开头到值为 nValue 的所有元素 (不包含 nValue)。要清空 set 和 multiset 的内容, 可使用其成员函数 clear()。

学习 set 和 multiset 的基本函数后, 接下来来看一个使用 set 容器的实际应用程序。程序清单 20.5 是基于菜单的电话簿的简单实现, 它让用户能够插入、查找、删除和显示人名和电话号码。

程序清单 20.5 一个使用 STL set 及其成员函数 find 和 erase 的电话簿

```
1: #include <set>
2: #include <iostream>
3: #include <string>
4:
5: using namespace std;
6:
7: enum MenuOptionSelection
8: {
9:     InsertContactsetEntry = 0,
10:    DisplayEntries = 1,
11:    FindNumber = 2,
12:    EraseEntry = 3,
13:    QuitApplication = 4
14: };
15:
16: struct ContactItem
17: {
18:     string strContactsName;
19:     string strPhoneNumber;
20:
21:     // Constructor
22:     ContactItem (const string& strName, const string & strNumber)
23:     {
24:         strContactsName = strName;
25:         strPhoneNumber = strNumber;
26:     }
27:
28:     bool operator == (const ContactItem& itemToCompare) const
29:     {
30:         return (itemToCompare.strContactsName == this->strContactsName);
31:     }
32:
33:     bool operator < (const ContactItem& itemToCompare) const
34:     {
35:         return (this->strContactsName < itemToCompare.strContactsName);
36:     }
37: };
38:
39: int ShowMenu ();
40: ContactItem GetContactInfo ();
41: void DisplayContactset (const set <ContactItem>& setContacts);
42: void FindContact (const set <ContactItem>& setContacts);
43: void EraseContact (set <ContactItem>& setContacts);
44:
45: int main ()
46: {
47:     set <ContactItem> setContacts;
48:     int nUserSelection = InsertContactsetEntry;
49:
50:     while ((nUserSelection = ShowMenu ()) != (int) QuitApplication)
51:     {
52:         switch (nUserSelection)
53:         {
54:             case InsertContactsetEntry:
55:                 setContacts.insert (GetContactInfo ());
56:                 cout << "Contacts set updated!" << endl << endl;
57:                 break;
58:
59:             case DisplayEntries:
60:                 DisplayContactset (setContacts);
61:                 break;
62:
63:             case FindNumber:
64:                 FindContact (setContacts);
65:                 break;
66:
67:             case EraseEntry:
68:                 EraseContact (setContacts);
69:                 DisplayContactset (setContacts);
70:                 break;
71:         }
```



```

72:         default:
73:             cout << "Invalid input '" << nUserSelection;
74:             cout << "' Please choose an option between 0 and 4" << endl;
75:             break;
76:         }
77:     }
78:
79:     cout << "Quitting! Bye!" << endl;
80:     return 0;
81: }
82:
83: void DisplayContactset (const set <ContactItem>& setContacts)
84: {
85:     cout << "*** Displaying contact information ***" << endl;
86:     cout << "There are " << setContacts.size () << " entries:" << endl;
87:
88:     set <ContactItem>::const_iterator iContact;
89:     for ( iContact = setContacts.begin ()
90:         ; iContact != setContacts.end ()
91:         ; ++ iContact )
92:         cout << "Name: '" << iContact->strContactsName << "' Number: '"
93:         << iContact->strPhoneNumber << "'" << endl;
94:
95:     cout << endl;
96: }
97:
98: ContactItem GetContactInfo ()
99: {
100:     cout << "*** Feed contact information ***" << endl;
101:     string strName;
102:     cout << "Please enter the person's name" << endl;;
103:     cout << "> ";
104:     cin >> strName;
105:
106:     string strPhoneNumber;
107:     cout << "Please enter "<< strName << "'s phone number" << endl;
108:     cout << "> ";
109:     cin >> strPhoneNumber;
110:
111:     return ContactItem (strName, strPhoneNumber);
112: }
113:
114: int ShowMenu ()
115: {
116:     cout << "*** What would you like to do next? ***" << endl << endl;
117:     cout << "Enter 0 to feed a name and phone number" << endl;
118:     cout << "Enter 1 to Display all entries" << endl;
119:     cout << "Enter 2 to find an entry" << endl;
120:     cout << "Enter 3 to erase an entry" << endl;
121:     cout << "Enter 4 to quit this application" << endl << endl;
122:     cout << "> ";
123:
124:     int nOptionSelected = 0;
125:
126:     // Accept user input
127:     cin >> nOptionSelected ;
128:     cout << endl;
129:     return nOptionSelected;
130: }
131:
132: void FindContact (const set <ContactItem>& setContacts)
133: {
134:     cout << "*** Find a contact ***" << endl;
135:     cout << "Whose number do you wish to find?" << endl;
136:     cout << "> ";
137:     string strName;
138:     cin >> strName;
139:
140:     set <ContactItem>::const_iterator iContactFound
141:         = setContacts.find (ContactItem (strName, ""));
142:
143:     if (iContactFound != setContacts.end ())
144:     {

```

```

145:         cout << strName << " is reachable at number: ";
146:         cout << iContactFound->strPhoneNumber << endl;
147:     }
148:     else
149:         cout << strName << " was not found in the contacts list" << endl;
150:
151:     cout << endl;
152:
153:     return;
154: }
155:
156: void EraseContact (set <ContactItem>& setContacts)
157: {
158:     cout << "*** Erase a contact ***" << endl;
159:     cout << "Whose number do you wish to erase?" << endl;
160:     cout << "> ";
161:     string strName;
162:     cin >> strName;
163:
164:     size_t nErased = setContacts.erase (ContactItem (strName, ""));
165:     if (nErased > 0)
166:         cout << strName << "'s contact information erased." << endl;
167:     else
168:         cout << strName << " was not found!" << endl;
169:
170:     cout << endl;
171: }

```

#### ▼ 输出:

\*\*\* What would you like to do next? \*\*\*

Enter 0 to feed a name and phone number  
Enter 1 to Display all entries  
Enter 2 to find an entry  
Enter 3 to erase an entry  
Enter 4 to quit this application

> 0

\*\*\* Feed contact information \*\*\*

Please enter the person's name  
> Tina  
Please enter Tina's phone number  
> 78966413  
Contacts set updated!

\*\*\* What would you like to do next? \*\*\*

Enter 0 to feed a name and phone number  
Enter 1 to Display all entries  
Enter 2 to find an entry  
Enter 3 to erase an entry  
Enter 4 to quit this application

> 0

\*\*\* Feed contact information \*\*\*

Please enter the person's name  
> Jack  
Please enter Jack's phone number  
> 456654213  
Contacts set updated!

\*\*\* What would you like to do next? \*\*\*

Enter 0 to feed a name and phone number  
Enter 1 to Display all entries  
Enter 2 to find an entry  
Enter 3 to erase an entry  
Enter 4 to quit this application

> 0

\*\*\* Feed contact information \*\*\*



```
Please enter the person's name
> Shawn
Please enter Shawn's phone number
> 77746123
Contacts set updated!

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to Display all entries
Enter 2 to find an entry
Enter 3 to erase an entry
Enter 4 to quit this application

> 0

*** Feed contact information ***
Please enter the person's name
> Fritz
Please enter Fritz's phone number
> 654666123
Contacts set updated!

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to Display all entries
Enter 2 to find an entry
Enter 3 to erase an entry
Enter 4 to quit this application

> 1
*** Displaying contact information ***
There are 4 entries:
Name: 'Fritz' Number: '654666123'
Name: 'Jack' Number: '456654213'
Name: 'Shawn' Number: '77746123'
Name: 'Tina' Number: '78966413'

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to Display all entries
Enter 2 to find an entry
Enter 3 to erase an entry
Enter 4 to quit this application .

> 2

*** Find a contact ***
Whose number do you wish to find?
> Shawn
Shawn is reachable at number: 77746123

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to Display all entries
Enter 2 to find an entry
Enter 3 to erase an entry
Enter 4 to quit this application

> 3

*** Erase a contact ***
Whose number do you wish to erase?
> Steve
Steve was not found!

*** Displaying contact information ***
There are 4 entries:
Name: 'Fritz' Number: '654666123'
Name: 'Jack' Number: '456654213'
Name: 'Shawn' Number: '77746123'
```



```
Name: 'Tina' Number: '78966413'

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to Display all entries
Enter 2 to find an entry
Enter 3 to erase an entry
Enter 4 to quit this application

> 3

*** Erase a contact ***
Whose number do you wish to erase?
> Jack
Jack's contact information erased.

*** Displaying contact information ***
There are 3 entries:
Name: 'Fritz' Number: '654666123'
Name: 'Shawn' Number: '77746123'
Name: 'Tina' Number: '78966413'

*** What would you like to do next? ***

Enter 0 to feed a name and phone number
Enter 1 to Display all entries
Enter 2 to find an entry
Enter 3 to erase an entry
Enter 4 to quit this application

> 4

Quitting! Bye!
```

#### ▼ 分析:

这个简单的电话簿实现基于一个 STL set，后者包含类型为 ContactItem 的对象。ContactItem 是一个结构，它是在第 16~37 行定义，包含两个属性：人名和电话号码。该结构定义的运算符<提供了默认的排序方式——根据人名进行排序。为了让 set 能够存储该结构的对象，这是必不可少的。因此，在显示存储的元素时，输出是按联系人姓名的字母顺序排列的。函数 FindContact（如第 132~154 行所示）演示了如何使用成员函数 set::find()。注意，find 操作的结果是一个迭代器，如第 140 行所示。首先将迭代器与集合末尾进行比较（如第 112 行所示），以判断 find 是否在 set 中找到了值。如果是，迭代器将指向 set 中的一个元素（ContactItem），因此使用它来显示电话号码，如第 146 行所示；否则，显示一条消息指出没有找到值，如第 149 行所示。注意，这种 find 语法适用于所有 STL 容器，其用法很简单。同样，第 156~171 行定义的 EraseContact 演示了如何使用成员函数 erase 将元素从 STL set 中删除。

#### 提示

这个电话簿实现是基于 STL set 的，因此不允许多个元素包含相同的值。如果要使电话簿中存储两个相同的人名（如 Tom），则应使用 STL multiset。如果 setContacts 为 multiset，上述代码可正确运行。要使用 multiset 存储多个值相同的元素，应使用 count() 成员函数来获悉有多少个元素包含特定的值，如程序清单 20.4 所示。这在前面的代码示例中演示过。在 multiset 中，类似的元素是相邻的，find 函数将返回指向第一个找到的元素的迭代器。可将该迭代器递增以获取下一个元素，递增次数为 count() 的返回值减 1。

## 20.3 使用 STL set 和 multiset 的优缺点

对需要频繁查找的应用程序来说，STL set 和 multiset 很有优势，因为其内容是经过排序的，因此查找元素的速度更快。然而，为提供这种优势，容器在插入元素时进行排序。因此，插入元素时将有额外开销，因为元素是经过排序的——如果应用程序将使用诸如 find() 等利用内部二叉树结构的函数，则这种开销是值得的。这种有序的二叉树结构使得 set 和 multiset 与顺序容器（如 vector）相比有一个

缺点：在 vector 中，可以使用新值替换迭代器（如 `std::find()` 返回的迭代器）指向的元素；但 set 根据元素的值对其进行了排序，因此不能使用迭代器覆盖元素的值，虽然通过编程可实现这种功能。

## 20.4 总结

本章介绍了 STL set 和 multiset 及其重要的成员函数和特征，还通过一个基于菜单的简单电话簿演示了如何使用 set 和 multiset，该电话簿提供了搜索和删除功能。

## 20.5 问与答

问：如何声明一个其元素按降序排列的整型 set？

答：set <int> 定义一个整型 set，这种 set 使用默认排序谓词 `std::less <T>` 将元素按升序排列，也可将其定义为 set <int, less <int> >。要按降序排列，应将 set 定义为 set <int, greater <int> >。

问：如果在一个字符串 set 中插入字符串 Jack 两次，将发生什么情况？

答：set 不能存储非唯一的值，因此 `std::set` 类的实现不允许插入相同的值。

问：在前一个例子中，如果需要两个 Jack，该如何办？

答：set 只能存储唯一的值。应选择使用 multiset。

问：multiset 的哪一个成员函数返回容器中有多少个元素包含特定的值？

答：count (value) 函数。

问：我使用 find 函数在 map 中找到元素，并有一个指向该元素的迭代器。能否使用这个迭代器来修改它指向的元素的值？

答：不能。有些 STL 实现可能允许用户通过迭代器（如 find 函数返回的迭代器）修改元素的值，但不应这样做。应将指向 set 中元素的迭代器视为 const 迭代器，即使 STL 实现没有强制这样做。

## 20.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 20.6.1 测验

1. 使用 set <int> 声明整型 set 时，排序标准将由哪个函数提供？
2. 在 multiset 中，重复的值以什么方式出现？
3. set 和 multiset 的哪个成员函数指出容器包含多少个元素？

### 20.6.2 练习

1. 在不修改 ContactItem 的情况下，扩展本章中的电话簿应用程序使其能够根据电话号码查找人名（提示：定义 set 时指定一个二元谓词，以便根据电话号码对元素进行排序，从而覆盖根据运算符 < 进行排序的默认方式）。
2. 定义一个 multiset 来存储单词及其含义，即将 multiset 用作词典（提示：multiset 存储的对象应是一个包含两个字符串的结构，其中一个字符串为单词，另一个字符串是单词的含义）。
3. 通过一个简单程序演示 set 不接受重复元素，而 multiset 接受。



## 第 21 章

# STL map 和 multimap

标准模板库 (STL) 向程序员提供了一些容器类，供需要进行频繁而快速搜索的应用程序使用。在本章中，您将学习：

- STL map 和 multimap 简介
- STL map 和 multimap 的基本操作
- 使用排序谓词定制排序行为

### 21.1 简介

map 和 multimap 是键-值对容器，支持根据键进行查找。map 和 multimap 之间的区别在于，后者能够存储重复的键，而前者只能存储唯一的键。

为实现快速查找，STL map 和 multimap 的内部结构看起来像一棵二叉树。这意味着在 map 或 multimap 中插入元素时将进行排序。这还意味着不像 vector 那样可以使用其他元素替换给定位置的元素，位于 map 中特定位置的元素不能替换为值不同的新元素，这是因为 map 将把新元素同二叉树中的其他元素进行比较，进而将其放在其他位置。

要使用 STL map 或 multimap 类，程序员必须包含头文件 <map>。

### 21.2 STL map 和 multimap 的基本操作

STL map 和 multimap 都是模板类，要使用其成员函数，必须先实例化。要使用 STL map 或 multimap 类，程序员必须包含头文件 <map>。

#### 21.2.1 实例化 std::map 对象

要实例化 map 或 multimap，必须具体化 std 命名空间中相应的模板类。实例化模板类 map 时，程序员需要指定键的类型、值的类型和可选的谓词（它帮助 map 类对插入的元素进行排序）。因此，典型的 map 实例化语法如下：

```
#include <map>
using namespace std;
...
map <keyType, valueType, Predicate=std::less <keyType> > mapObject;
multimap <keyType, valueType, Predicate=std::less <keyType> > mmapObject;
```

程序清单 21.1 更详细地说明了这种语法。

程序清单 21.1 实例化 STL map 和 multimap（键类型为整型，值类型为字符串）

```
1: #include <map>
2: #include <string>
3:
```

```
4: int main ()
5: {
6:     using namespace std;
7:
8:     map <int, string> mapIntegersToString;
9:     multimap <int, string> mmapIntegersToString;
10:
11:     return 0;
12: }
```

### ▼ 分析:

第1行指出要使用 STL 类 map 和 multimap, 必须包含标准头文件, 该文件位于 std 命名空间内。第8行和第9行演示了如何实例化这两个模板类, 其中键类型为 int, 值类型为 string。

### 提示

实例化模板类 map 时, 还需指定第3个参数, 它指定了排序标准。在前面的例子中, 没有指定第3个参数, 因此使用默认排序标准, 即基于 STL 函数 less 进行排序, 该函数使用 < 运算符来比较两个对象。如果要覆盖这种默认排序机制, 应在实例化 map 的语法中提供第3个参数——一个二元谓词, 如下所示:

```
map <KeyType, ValueType, BinaryPredicate>
mapWithCustomSortPredicate;
```

## 21.2.2 在 STL map 或 multimap 中插入元素

map 和 multimap 的大多数函数的用法类似, 它们接受类似的参数, 返回类型也类似。例如, 要在这两种容器中插入元素, 都可使用成员函数 insert, 如程序清单 21.2 所示。

程序清单 21.2 在 STL map 或 multimap 中插入元素

```
1: #include <map>
2: #include <iostream>
3:
4: using namespace std;
5:
6: // Type-define the map and multimap definition for easy readability
7: typedef map <int, string> MAP_INT_STRING;
8: typedef multimap <int, string> MMAP_INT_STRING;
9:
10: int main ()
11: {
12:     MAP_INT_STRING mapIntToString;
13:
14:     // Insert key-value pairs into the map using value_type
15:     mapIntToString.insert (MAP_INT_STRING::value_type (3, "Three"));
16:
17:     // Insert a pair using function make_pair
18:     mapIntToString.insert (make_pair (-1, "Minus One"));
19:
20:     // Insert a pair object directly
21:     mapIntToString.insert (pair <int, string> (1000, "One Thousand"));
22:
23:     // Insert using an array-like syntax for inserting key-value pairs
24:     mapIntToString [1000000] = "One Million";
25:
26:     cout << "The map contains " << mapIntToString.size ();
27:     cout << " key-value pairs. " << endl;
28:     cout << "The elements in the map are: " << endl;
29:
30:     // Print the contents of the map to the screen
31:     MAP_INT_STRING::const_iterator iMapPairLocator;
32:
33:     for ( iMapPairLocator = mapIntToString.begin ()
34:           ; iMapPairLocator != mapIntToString.end ()
35:           ; ++ iMapPairLocator )
36:     {
```

```

37:     cout << "Key: " << iMapPairLocator->first;
38:     cout << " Value: " << iMapPairLocator->second.c_str ();
39:
40:     cout << endl;
41: }
42:
43: MMAP_INT_STRING mmapIntToString;
44:
45: // The insert function works the same way for multimap too
46: mmapIntToString.insert (MMAP_INT_STRING::value_type (3, "Three"));
47: mmapIntToString.insert (MMAP_INT_STRING::value_type (45, "Forty Five"));
48: mmapIntToString.insert (make_pair (-1, "Minus One"));
49: mmapIntToString.insert (pair <int, string> (1000, "One Thousand"));
50:
51: // A multimap can store duplicates - insert one
52: mmapIntToString.insert (MMAP_INT_STRING::value_type (1000, "Thousand"));
53:
54: cout << endl << "The multimap contains " << mmapIntToString.size ();
55: cout << " key-value pairs." << endl;
56: cout << "The elements in the multimap are: " << endl;
57:
58: // Print the contents of the map to the screen
59: MMAP_INT_STRING::const_iterator iMultiMapPairLocator;
60:
61: for ( iMultiMapPairLocator = mmapIntToString.begin ()
62:       ; iMultiMapPairLocator != mmapIntToString.end ()
63:       ; ++ iMultiMapPairLocator )
64: {
65:     cout << "Key: " << iMultiMapPairLocator->first;
66:     cout << " Value: " << iMultiMapPairLocator->second.c_str ();
67:
68:     cout << endl;
69: }
70:
71: cout << endl;
72:
73: // The multimap can also return the number of pairs with the same key
74: cout << "The number of pairs in the multimap with 1000 as their key: "
75:     << mmapIntToString.count (1000) << endl;
76:
77: return 0;
78: }

```

### ▼ 输出:

```

The map contains 4 key-value pairs.
The elements in the map are:
Key: -1 Value: Minus One
Key: 3 Value: Three
Key: 1000 Value: One Thousand
Key: 1000000 Value: One Million

```

```

The multimap contains 5 key-value pairs.
The elements in the multimap are:
Key: -1 Value: Minus One
Key: 3 Value: Three
Key: 45 Value: Forty Five
Key: 1000 Value: One Thousand
Key: 1000 Value: One Thousand

```

```

The number of pairs in the multimap with 1000 as their key are: 2

```

### ▼ 分析:

在这个例子中，第 7 行和第 8 行给模板类 map 和 multimap 的实例指定别名。第 14~24 行演示了各种将元素插入到 map 中的方法。在该示例演示的 4 种方法中，只有最后一种方法是 map 特有的，它使用数组语法通过下标运算符（[]）来插入元素；其他 3 种方法也可使用于 multimap。另外，可使用迭代器访问 map 或 multimap 中的元素，如第 29~37 行所示。迭代器指向容器中的一个元素，如果容器是 map 或 multimap，则元素是一个键-值对，因此迭代器指向一个键-值对。可通过成员 first 和 second

访问键-值对的内容，如第 37 行和第 38 行所示。其中前者是键，后者是值，输出也说明了这一点。

### 21.2.3 在 STL map 或 multimap 中查找元素

诸如 map 和 multimap 等关联容器都提供了成员函数 find()，它让您能够根据给定的键来查找值。程序清单 21.3 演示了 multimap::find 的用法。

程序清单 21.3 使用 multimap 的成员函数 find

```

1: #include <map>
2: #include <iostream>
3: #include <string>
4:
5: using namespace std;
6:
7: // Typedef the multimap definition for easy readability
8: typedef multimap <int, string> MMAP_INT_STRING;
9:
10: int main ()
11: {
12:     MMAP_INT_STRING mmapIntToString;
13:
14:     // The insert function works the same way for multimap too
15:     mmapIntToString.insert (MMAP_INT_STRING::value_type (3, "Three"));
16:     mmapIntToString.insert (MMAP_INT_STRING::value_type (45, "Forty Five"));
17:     mmapIntToString.insert (MMAP_INT_STRING::value_type (-1, "Minus One"));
18:     mmapIntToString.insert (MMAP_INT_STRING::value_type (1000, "Thousand"));
19:
20:     // A multimap can store duplicates - insert one
21:     mmapIntToString.insert (MMAP_INT_STRING::value_type
22:                             (1000, "Thousand (duplicate)"));
23:
24:     cout << "The multimap contains " << mmapIntToString.size ();
25:     cout << " key-value pairs." << endl;
26:     cout << "The elements in the multimap are: " << endl;
27:
28:     // Print the contents of the map to the screen
29:     MMAP_INT_STRING::const_iterator iMultiMapPairLocator;
30:
31:     for ( iMultiMapPairLocator = mmapIntToString.begin ()
32:           ; iMultiMapPairLocator != mmapIntToString.end ()
33:           ; ++ iMultiMapPairLocator )
34:     {
35:         cout << "Key: " << iMultiMapPairLocator->first;
36:         cout << ", Value: " << iMultiMapPairLocator->second << endl;
37:     }
38:
39:     cout << endl;
40:
41:     cout << "Finding all key-value pairs with 1000 as their key: " << endl;
42:
43:     // Find an element in the multimap using the 'find' function
44:     MMAP_INT_STRING::const_iterator iElementFound;
45:
46:     iElementFound = mmapIntToString.find (1000);
47:
48:     // Check if "find" succeeded
49:     if (iElementFound != mmapIntToString.end ())
50:     {
51:         // Find the number of pairs that have the same supplied key
52:         size_t nNumPairsInMap = mmapIntToString.count (1000);
53:         cout << "The number of pairs in the multimap with 1000 as key: ";
54:         cout << nNumPairsInMap << endl;
55:
56:         // Output those values to the screen
57:         cout << "The values corresponding to the key 1000 are: " << endl;
58:         for ( size_t nValuesCounter = 0
59:               ; nValuesCounter < nNumPairsInMap
60:               ; ++ nValuesCounter )

```

```

61:         {
62:             cout << "Key: " << iElementFound->first;
63:             cout << ", Value [" << nValuesCounter << "] = ";
64:             cout << iElementFound->second << endl;
65:
66:             ++ iElementFound;
67:         }
68:     }
69:     else
70:         cout << "Element not found in the multimap";
71:
72:     return 0;
73: }

```

#### ▼ 输出:

```

The multimap contains 5 key-value pairs.
The elements in the multimap are:
Key: -1, Value: Minus One
Key: 3, Value: Three
Key: 45, Value: Forty Five
Key: 1000, Value: Thousand
Key: 1000, Value: Thousand (duplicate)

Finding all pairs with 1000 as their key...
The number of pairs in the multimap with 1000 as key: 2
The values corresponding to the key 1000 are:
Key: 1000, Value [0] = Thousand
Key: 1000, Value [1] = Thousand (duplicate)

```

#### ▼ 分析:

实例化 multimap 及插入元素的语法如第 1~21 行所示, 这在程序清单 21.1 和 21.2 分析过。第 46 行演示了成员函数 find 的用法。在 STL 中, find 总是返回一个迭代器, 它指向找到的元素 (这里是一个键-值对)。为判断 find 操作是否成功, 首先将这个迭代器与成员 end() 返回的迭代器进行比较, 如第 49 行所示。第 52~67 行演示了如何访问和显示 multimap 中键相同的元素。第 52 行使用 count() 确定容器中包含多少个键相同的键-值对。由于 find 返回的迭代器指向找到的第一个元素, 而 multimap 将值相同的所有元素放在相邻的位置, 因此只需向前移动迭代器就可访问其他包含相同键的键-值对。

## 21.2.4 删除 STL map 或 multimap 中的元素

map 和 multimap 提供了成员函数 erase(), 该函数删除容器中的元素。调用 erase 函数时将键作为参数, 这将删除所有包含指定键的元素:

```
mapObject.erase (key);
```

erase 函数的另一种版本接受迭代器作为参数, 并删除迭代器指向的元素:

```
mapObject.erase (iElement);
```

还可使用迭代器指定边界, 从而将指定范围内的所有元素都从 map 或 multimap 中删除:

```
mapObject.erase (iLowerBound, iUpperBound);
```

程序清单 21.4 演示了 erase() 函数的这些版本的用法。

#### 程序清单 21.4 删除 multimap 中的元素

```

1: #include <map>
2: #include <iostream>
3: #include <string>
4:
5: using namespace std;
6:
7: // typedef the multimap definition for easy readability
8: typedef multimap <int, string> MULTIMAP_INT_STRING;
9:
10: int main ()

```



```

11: {
12:     MULTIMAP_INT_STRING mmapIntToString;
13:
14:     // Insert key-value pairs into the multimap
15:     mmapIntToString.insert (MULTIMAP_INT_STRING::value_type (3, "Three"));
16:     mmapIntToString.insert (MULTIMAP_INT_STRING::value_type(45, "Forty
Five"));
17:     mmapIntToString.insert (MULTIMAP_INT_STRING::value_type (-1, "Minus
One"));
18:     mmapIntToString.insert (MULTIMAP_INT_STRING::value_type (1000,
"Thousand"));
19:
20:     // Insert duplicates into the multimap
21:     mmapIntToString.insert (MULTIMAP_INT_STRING::value_type (-1, "Minus
One"));
22:     mmapIntToString.insert (MULTIMAP_INT_STRING::value_type (1000,
"Thousand"));
23:
24:     cout << "The multimap contains " << mmapIntToString.size ();
25:     cout << " key-value pairs. " << "They are: " << endl;
26:
27:     // Print the contents of the multimap to the screen
28:     MULTIMAP_INT_STRING::const_iterator iPairLocator;
29:
30:     for ( iPairLocator = mmapIntToString.begin ()
31:           ; iPairLocator != mmapIntToString.end ()
32:           ; ++ iPairLocator )
33:     {
34:         cout << "Key: " << iPairLocator->first;
35:         cout << ", Value: " << iPairLocator->second.c_str () << endl;
36:     }
37:
38:     cout << endl;
39:
40:     // Erasing an element with key as -1 from the multimap
41:     if (mmapIntToString.erase (-1) > 0)
42:         cout << "Erased all pairs with -1 as key." << endl;
43:
44:     // Erase an element given an iterator from the multimap
45:     MULTIMAP_INT_STRING::iterator iElementLocator =
mmapIntToString.find(45);
46:     if (iElementLocator != mmapIntToString.end ())
47:     {
48:         mmapIntToString.erase (iElementLocator);
49:         cout << "Erased a pair with 45 as key using an iterator" << endl;
50:     }
51:
52:     // Erase a range from the multimap...
53:     cout << "Erasing the range of pairs with 1000 as key." << endl;
54:     mmapIntToString.erase ( mmapIntToString.lower_bound (1000)
55:                             , mmapIntToString.upper_bound (1000) );
56:
57:     cout << endl;
58:     cout << "The multimap now contains " << mmapIntToString.size ();
59:     cout << " key-value pair(s)." << "They are: " << endl;
60:
61:     // Print the contents of the multimap to the screen
62:     for ( iPairLocator = mmapIntToString.begin ()
63:           ; iPairLocator != mmapIntToString.end ()
64:           ; ++ iPairLocator )
65:     {
66:         cout << "Key: " << iPairLocator->first;
67:         cout << ", Value: " << iPairLocator->second.c_str () << endl;
68:     }
69:
70:     return 0;
71: }

```

#### ▼ 输出:

The multimap contains 6 key-value pairs. They are:  
Key: -1, Value: Minus One

```
Key: -1, Value: Minus One
Key: 3, Value: Three
Key: 45, Value: Forty Five
Key: 1000, Value: Thousand
Key: 1000, Value: Thousand
```

```
Erased all pairs with -1 as key.
Erased a pair with 45 as key using an iterator
Erasing the range of pairs with 1000 as key.
```

```
The multimap now contains 1 key-value pair(s).They are:
Key: 3, Value: Three
```

### ▼ 分析:

第 15~22 行将值插入到 multimap 中, 其中一些值是重复的 (因为 multimap 不同于 map, 它允许插入重复的元素)。将键-值对插入到 multimap 后, 第 41 行调用接受一个键作为参数的 erase 函数, 将所有包含该键 (-1) 的键-值对都删除。第 45 行调用 multimap::find 函数在 map 中查找包含键 45 的键-值对。第 48 行使用 find 操作返回的迭代器将其指向的键-值对从 multimap 中删除。最后, 第 54 行将一系列包含键 1 000 的元素删除。这个范围是由函数 lower\_bound 和 upper\_bound 指定的, 从输出可知, 这种 erase 操作导致 map 中所有键为 1 000 的键-值对都被删除。

### 提示

在程序清单 21.3 中, 用于遍历 map 中元素的迭代器 iPairLocator 的类型为 const\_iterator, 而 find 操作返回并传递给函数 erase 的迭代器 iElementFound 的类型不是 const\_iterator。这是有意为之的, 用于遍历容器并显示元素的迭代器不需要修改容器, 因此其类型应为 const\_iterator; 而用于 erase 等操作的迭代器需要修改容器的内容, 因此其类型不能是 const\_iterator。

## 21.3 提供自定义的排序谓词

map 和 multimap 的模板定义包含第 3 个参数, 该参数是确保 map 能够正常工作的排序谓词。如果没有指定这个参数 (如前面的示例所示), 将使用 std::less <> 提供的默认排序标准, 该谓词使用 < 运算符来比较两个对象。

然而, 在有些情况下, 可能需要根据其他标准对 map 进行排序和搜索。对于键类型为 std::string 的 map, 默认排序标准是 std::string 类定义的 < 运算符, 因此区分大小写。很多应用程序 (如电话簿) 要求执行插入和搜索操作时不区分大小写, 为满足这种需求, 一种解决方案是在实例化 map 时提供一个排序谓词, 它根据不区分大小写的比较结果返回 true 或 false, 如下所示:

```
map <keyType, valueType, Predicate> mapObject;
```

程序清单 21.5 对此做了更详细的解释。

程序清单 21.5 编写电话簿应用程序时, 在实例化 std::map 时提供一个自定义排序谓词

```
1: #include <map>
2: #include <algorithm>
3: #include <string>
4: #include <iostream>
5:
6: using namespace std;
7:
8: /*
9: This is the binary predicate that helps the map sort
10: string-keys irrespective of their case
11: */
12: struct CCaseInsensitive
13: {
14:     bool operator () (const string& str1, const string& str2) const
15:     {
16:         string str1NoCase (str1), str2NoCase (str2);
17:         transform (str1.begin(), str1.end(), str1NoCase.begin(), tolower);
18:         transform (str2.begin(), str2.end(), str2NoCase.begin(), tolower);
19:
```

```

20:         return (str1NoCase < str2NoCase);
21:     };
22: };
23:
24:
25: // Typedef map definitions for easy readability...
26: // A directory that sorts keys using string::operator < (case sensitive)
27: typedef map <string, string> DIRECTORY_WITHCASE;
28:
29: // A case-insensitive directory definition
30: typedef map <string, string, CCaseInsensitive> DIRECTORY_NOCASE;
31:
32: int main ()
33: {
34:     // Case-insensitive directory: case of the string-key plays no role
35:     DIRECTORY_NOCASE dirNoCase;
36:
37:     dirNoCase.insert (DIRECTORY_NOCASE::value_type ("John", "2345764"));
38:     dirNoCase.insert (DIRECTORY_NOCASE::value_type ("JOHN", "2345764"));
39:     dirNoCase.insert (DIRECTORY_NOCASE::value_type ("Sara", "42367236"));
40:     dirNoCase.insert (DIRECTORY_NOCASE::value_type ("Jack", "32435348"));
41:
42:     cout << "Displaying contents of the case-insensitive map:" << endl;
43:
44:     // Print the contents of the map to the screen
45:     DIRECTORY_NOCASE::const_iterator iPairLocator1;
46:     for ( iPairLocator1 = dirNoCase.begin()
47:           ; iPairLocator1 != dirNoCase.end()
48:           ; ++ iPairLocator1 )
49:     {
50:         cout << "Name: " << iPairLocator1->first;
51:         cout << ", Phone number: " << iPairLocator1->second << endl;
52:     }
53:
54:     cout << endl;
55:
56:     // Case-sensitive directory: case of the string-key affects
57:     // insertion & search
58:     DIRECTORY_WITHCASE dirWithCase;
59:
60:     // Take sample values from previous map...
61:     dirWithCase.insert ( dirNoCase.begin(), dirNoCase.end() );
62:
63:     cout << "Displaying contents of the case-sensitive map:" << endl;
64:
65:     // Print the contents of the map to the screen
66:     DIRECTORY_WITHCASE::const_iterator iPairLocator2;
67:     for ( iPairLocator2 = dirWithCase.begin()
68:           ; iPairLocator2 != dirWithCase.end()
69:           ; ++ iPairLocator2 )
70:     {
71:         cout << "Name: " << iPairLocator2->first;
72:         cout << ", Phone number: " << iPairLocator2->second << endl;
73:     }
74:
75:     cout << endl;
76:
77:     // Search for a name in the two maps and display result
78:     cout << "Please enter a name to search: " << endl << "> ";
79:     string strNameInput;
80:     cin >> strNameInput;
81:
82:     DIRECTORY_NOCASE::const_iterator iSearchResult1;
83:
84:     // find in the map...
85:     iSearchResult1 = dirNoCase.find (strNameInput);
86:     if (iSearchResult1 != dirNoCase.end())
87:     {
88:         cout<<iSearchResult1->first<< "'s number in the case-insensitive";
89:         cout << " directory is: " << iSearchResult1->second << endl;
90:     }
91:     else
92:     {

```

```

93:         cout << strNameInput << "'s number not found ";
94:         cout << "in the case-insensitive directory" << endl;
95:     }
96:
97:     DIRECTORY_WITHCASE::const_iterator iSearchResult2;
98:
99:     // find in the case-sensitive map...
100:    iSearchResult2 = dirWithCase.find (strNameInput);
101:    if (iSearchResult2 != dirWithCase.end())
102:    {
103:        cout<< iSearchResult2->first<< "'s number in the case-sensitive";
104:        cout << " directory is: " << iSearchResult2->second << endl;
105:    }
106:    else
107:    {
108:        cout << strNameInput << "'s number was not found ";
109:        cout << "in the case-sensitive directory" << endl;
110:    }
111:
112:    return 0;
113: }

```

### ▼ 输出:

```

Displaying the contents of the case-insensitive map on the screen...
Name: Jack, Phone number: 32435348
Name: John, Phone number: 2345764
Name: Sara, Phone number: 42367236

Displaying the contents of the case-sensitive map on the screen...
Name: Jack, Phone number: 32435348
Name: John, Phone number: 2345764
Name: Sara, Phone number: 42367236

Please enter a name to search in the directories:
> JOHN
John's number from the case-insensitive directory is: 2345764
JOHN's number was not found in the case-sensitive directory

```

### ▼ 分析:

程序清单 21.5 基于 map 实现了两个电话簿，其中 map 的键和值的类型都是字符串。其中一个电话簿是区分大小写的，定义它是没有提供谓词（如第 27 行所示），因此使用默认谓词（即 `std::less`），该谓词调用键（即 `std::string` 类）的运算符 `<`，该运算符区分大小写。定义另一个 map 时提供了谓词 `CcaseInsensitive`，该谓词在对两个字符串进行比较时不考虑大小写，并在第一个字符串小于第二个字符串时返回 `true`。从第 14~21 行可知，这个二元谓词接受两个字符串作为输入，将它们都转换为小写，然后再进行比较，最后返回一个布尔值。将这个谓词提供给 map 时，map 将在插入和搜索元素时不考虑大小写。从输出可知，这两个电话簿对象包含的元素完全相同，但用户输入在这两个电话簿中都有（但大小写不同）的人名（即要搜索的键）时，能够在不区分大小写的电话簿中找到它（这要归功于谓词函数对象），但在区分大小写电话簿中却找不到。

这个示例演示了如何使用谓词来定制 map 的行为，它还表明键可以是任何类型的，而程序员可提供一个谓词以定义这种类型的 map 的行为。注意，这里使用的谓词是一个实现了运算符 `()` 的结构，但它也可以是一个类。

这种也可以作为函数的对象被称为函数对象（或 Functor），函数对象将在第 22 章更详细地介绍。

## 21.4 总结

本章介绍了 STL map 和 multimap 的用法及其重要的成员函数和特征。还阐述了使用谓词来定制排序标准的重要性，如程序清单 21.5 的电话簿应用程序所示。

## 21.5 问与答

问：如何声明一个其元素按降序排列的整型 set？

答: map <int> 定义一个整型 map, 这种 map 使用默认排序谓词 std::less <T> 将元素按升序排列, 也可将其定义为 map <int, less <int> >。要按降序排列, 应将 map 定义为 map <int, greater <int> >。

问: 如果在一个字符串 map 中插入字符串 Jack 两次, 将发生什么情况?

答: map 不能存储非唯一的值, 因此 std::map 类的实现不允许插入相同的值。

问: 在前一个例子中, 如果需要两个 Jack, 该如何办?

答: map 只能存储唯一的值, 应选择使用 multimap。

问: multimap 的哪一个成员函数返回容器中有多少个元素包含特定的值?

答: count (value) 函数。

问: 我使用 find 函数在 map 中找到了一个元素, 并有一个指向该元素的迭代器。能否使用这个迭代器来修改它指向的元素的值?

答: 不能。有些 STL 实现可能允许用户通过迭代器 (如 find 函数返回的迭代器) 修改元素的值, 但不应这样做。应将指向 map 中元素的迭代器视为 const 迭代器, 即使 STL 实现没有强制这样做。

## 21.6 作业

作业包括测验和练习, 前者帮助加深读者对所学知识的理解, 后者提供了使用新学知识的机会。请尽量先完成测验和练习题, 然后再对照附录 D 的答案, 继续学习下一章前, 请务必弄懂这些答案。

### 21.6.1 测验

1. 使用 map <int> 声明整型 map 时, 排序标准将由哪个函数提供?
2. 在 multimap 中, 重复的值以什么方式出现?
3. map 和 multimap 的哪个成员函数指出容器包含多少个元素?
4. 在 map 中的什么地方可以找到重复的值?

### 21.6.2 练习

1. 编写一个应用程序来实现电话簿, 它不要求人名是唯一的。应选择哪种容器? 写出容器的定义。
2. 下面是电话簿应用程序中一个 map 的定义:

```
map <wordProperty, string, fPredicate> mapWordDefinition;
```

其中 wordProperty 是一个结构:

```
struct wordProperty
{
    string strWord;
    bool bIsFromLatin;
};
```

请定义二元谓词 fPredicate, 用于帮助该 map 根据类型为 wordProperty 的键包含的 string 属性对元素进行排序。

3. 通过一个简单程序演示 map 不接受重复元素, 而 multimap 接受。





## 第四部分

### 再谈 STL

第 22 章 理解函数对象

第 23 章 STL 算法

第 24 章 自适应容器：栈和队列

第 25 章 使用 STL 位标志

## 第 22 章

# 理解函数对象

函数对象（也称 functor）听起来陌生或难以理解，但它们是 C++ 实体，即使您没有用过，也很可能见过，只是您没有意识到而已。

在本章中，您将学习：

- 函数对象的概念
- 将函数对象用作谓词
- 如何使用函数对象实现一元和二元谓词

### 22.1 函数对象与谓词的概念

从概念上说，函数对象是用作函数的对象；但从实现上说，函数对象是实现了 `operator()` 的类的对象。虽然函数和函数指针也可归为函数对象，但实现了 `operator()` 的类的对象才能保存状态（即类的成员属性的值），才能用于标准模板库（STL）算法。

C++ 程序员常用于 STL 算法的函数对象可分为下列两种类型。

- 一元函数：接受一个参数的函数，如  $f(x)$ 。如果一元函数返回一个布尔值，则该函数称为谓词。
- 二元函数：接受两个参数的函数，如  $f(x, y)$ 。如果二元函数返回一个布尔值，则该函数称为二元谓词。

返回布尔类型的函数对象通常用于需要进行判断的算法。组合两个函数对象的函数对象称为自适应函数对象。

### 22.2 函数对象的典型用途

可以通过长篇大论从理论上解释函数对象，也可通过小型应用程序看看函数对象是什么样的及其工作原理。下面将采取后一种实用方法，直接看看如何在 C++ 编程中使用函数对象。

#### 22.2.1 一元函数

只对一个参数进行操作的函数称为一元函数。一元函数的功能可能很简单，如在屏幕上显示元素，如下所示：

```
// A unary function
template <typename elementType>
void FuncDisplayElement (const elementType & element)
{
    cout << element << ' ';
};
```

函数 `FuncDisplayElement` 接受一个类型为模板化类型 `elementType` 的参数，并使用控制台输出语句 `std::cout` 将该参数显示出来。该函数也可采用另一种表现形式，即其实现包含在类或结构的 `operator()` 中：

```
// Struct that can behave as a unary function
template <typename elementType>
struct DisplayElement
{
    void operator () (const elementType& element) const
    {
        cout << element << ' ';
    }
};
```

**提示**

DisplayElement 是一个结构, 如果它是类, 则必须给 operator() 指定访问权限修饰符 public。  
结构相当于成员默认为公有的类。

这两种实现都可用于 STL 算法 for\_each, 将集合中的内容显示在屏幕上, 每次显示一个元素, 如程序清单 22.1 所示。

**程序清单 22.1 使用一元函数将集合的内容显示在屏幕上**

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <list>
5:
6: using namespace std;
7:
8: // struct that behaves as a unary function
9: template <typename elementType>
10: struct DisplayElement
11: {
12:     void operator () (const elementType& element) const
13:     {
14:         cout << element << ' ';
15:     }
16: };
17:
18: int main ()
19: {
20:     vector <int> vecIntegers;
21:
22:     for (int nCount = 0; nCount < 10; ++ nCount)
23:         vecIntegers.push_back (nCount);
24:
25:     list <char> listChars;
26:
27:     for (char nChar = 'a'; nChar < 'k'; ++nChar)
28:         listChars.push_back (nChar);
29:
30:     cout << "Displaying the vector of integers: " << endl;
31:
32:     // Display the array of integers
33:     for_each ( vecIntegers.begin ()      // Start of range
34:              , vecIntegers.end ()        // End of range
35:              , DisplayElement <int> () ); // Unary function object
36:
37:     cout << endl << endl;
38:     cout << "Displaying the list of characters: " << endl;
39:
40:     // Display the list of characters
41:     for_each ( listChars.begin ()        // Start of range
42:              , listChars.end ()          // End of range
43:              , DisplayElement <char> () ); // Unary function object
44:
45:     return 0;
46: }
```

**▼ 输出:**

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9
```

```
Displaying the list of characters:  
a b c d e f g h i j
```

### ▼ 分析:

第9~16行包含函数对象 `DisplayElement`，它实现了 `operator()`。在第33~35行，将这个函数对象用于了 STL 算法 `std::for_each`。`for_each` 接受3个参数：第1个参数指定范围的起点，第2个参数指定范围的终点，第3个参数是要对指定范围内的每个元素调用的函数。换句话说，这些代码将对向量 `verIntegers` 中的每个元素调用 `DisplayElement::operator()`。注意，在这里可不使用结构 `DisplayElement`，而使用 `FuncDisplayElement`，其效果相同。第40~43行显示了字符 `list` 的内容。

如果能够使用结构的对象来存储信息，则使用在结构中实现的函数对象的优点将显现出来。这是 `FuncDisplayElement` 不像结构那么强大的地方，因为结构除 `operator()` 外还可以有成员属性。下面是一个稍做修改的版本，它使用了成员属性：

```
template <typename elementType>  
struct DisplayElementKeepCount  
{  
    int m_nCount;  
  
    DisplayElementKeepCount () // constructor  
    {  
        m_nCount = 0;  
    }  
  
    void operator () (const elementType& element)  
    {  
        ++ m_nCount;  
        cout << element << ' ' ;  
    }  
};
```

在上述代码中，`DisplayElementKeepCount` 对前一个版本稍做了修改。`operator()` 不再是 `const` 成员函数，因为它对成员 `m_nCount` 进行递增（修改），以记录自己被调用用于显示数据的次数。该计数是通过公有成员属性 `m_nCount` 暴露的。程序清单 22.2 说明了使用可保存状态的函数对象的优点。

### 程序清单 22.2 使用函数对象存储状态

```
1: #include <algorithm>  
2: #include <iostream>  
3: #include <vector>  
4: #include <list>  
5:  
6: using namespace std;  
7:  
8: template <typename elementType>  
9: struct DisplayElementKeepCount  
10: {  
11:     // Hold the count in a member variable  
12:     int m_nCount;  
13:  
14:     // Constructor  
15:     DisplayElementKeepCount ()  
16:     {  
17:         m_nCount = 0;  
18:     }  
19:  
20:     // Display the element, hold count!  
21:     void operator () (const elementType& element)  
22:     {  
23:         ++ m_nCount;  
24:         cout << element << ' ' ;  
25:     }  
26: };  
27:  
28: int main ()  
29: {
```

```

30:     vector<int> vecIntegers;
31:
32:     for (int nCount = 0; nCount < 10; ++ nCount)
33:         vecIntegers.push_back (nCount);
34:
35:     cout << "Displaying the vector of integers: " << endl;
36:
37:     // Display the array of integers
38:     DisplayElementKeepCount<int> mResult;
39:     mResult = for_each ( vecIntegers.begin ()    // Start of range
40:                        , vecIntegers.end ()      // End of range
41:                        , DisplayElementKeepCount<int> () ); // function object
42:
43:     cout << endl << endl;
44:
45:     // Use the state stores in the return value of for_each!
46:     cout << "' ' << mResult.m_nCount << "' elements were displayed!" << endl;
47:
48:     return 0;
49: }

```

### ▼ 输出:

```

Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9

'10' elements were displayed!

```

### ▼ 分析:

这个例子与程序清单 22.1 所示示例的最大区别在于，将 `DisplayElementKeepCount` 用作 `for_each` 的返回值。对每个元素调用 `operator()` 时，`operator()` 显示该元素并递增内部计数，它将使用对象的次数存储在 `m_nCount` 中。在 `for_each` 执行完毕后，第 46 行使用这个对象指出显示了多少个元素。注意，在这种情况下，如果使用普通函数而不是在结构中实现的函数，将无法以如此直接的方式提供这种功能。

## 22.2.2 一元谓词

返回布尔值的一元函数是谓词。这种函数可供 STL 算法用于判断。程序清单 22.3 所示的谓词判断输入元素是否为初始值的整数倍。

程序清单 22.3 判断一个数字是否为另一个数字的整数倍的一元谓词

```

1: // A structure as a unary predicate
2: template<typename numberType>
3: struct IsMultiple
4: {
5:     numberType m_Divisor;
6:
7:     // divisorialize the divisor
8:     IsMultiple (const numberType& divisor)
9:     {
10:         m_Divisor = divisor;
11:     }
12:
13:     // The comparator of type: bool f(x)
14:     bool operator () (const numberType& element) const
15:     {
16:         // Check if the dividend is a multiple of the divisor
17:         return ((element % m_Divisor) == 0);
18:     }
19: };

```

### ▼ 分析:

这里的 `operator()` 返回布尔值，并用作一元谓词。该结构有一个构造函数，它初始化除数的值。然后



用保存在对象中的这个值来判断要比较的元素是否可以被它整除，如 `operator()` 的实现所示，它使用数学运算取模`%`来返回除法运算的余数。然后谓词将余数与零进行比较，以判断被除数是否为除数的整数倍。

在程序清单 22.4 中，使用程序清单 22.3 所示的谓词来判断集合中的数是否为 4 的整数倍。

程序清单 22.4 使用一元谓词 `IsMultiple`

---

```

1: #include <algorithm>
2: #include <vector>
3: #include <iostream>
4:
5: using namespace std;
6:
7: // Insert definition of struct IsMultiple from Listing 22.3 here
8:
9: int main ()
10: {
11:     vector<int> vecIntegers;
12:
13:     cout << "The vector contains the following sample values: ";
14:
15:     // Insert sample values: 25 - 31
16:     for (int nCount = 25; nCount < 32; ++ nCount)
17:     {
18:         vecIntegers.push_back (nCount);
19:         cout << nCount << ' ';
20:     }
21:
22:     cout << endl;
23:
24:     // Find the first element that is a multiple of 4 in the collection
25:     vector<int>::iterator iElement;
26:     iElement = find_if ( vecIntegers.begin ()
27:         , vecIntegers.end ()
28:         , IsMultiple<int> (4) ); // Unary predicate initialized to 4
29:
30:     if (iElement != vecIntegers.end ())
31:     {
32:         cout << "The first element in the vector divisible by 4 is: ";
33:         cout << *iElement << endl;
34:     }
35:
36:     return 0;
37: }
```

---

#### ▼ 输出:

---

```

The vector contains the following sample values: 25 26 27 28 29 30 31
The first element in the vector that is divisible by 4 is: 28
```

---

#### ▼ 分析:

这个例子首先声明了一个整型 `vector`，第 16~20 行将一些值插入到该容器中。第 26~28 行的 `find_if` 使用了一元谓词。在这里，将函数对象 `IsMultiple` 初始化为除数 4，`find_if` 对指定范围内的每个元素调用一元谓词 `IsMultiple::operator()`。当 `operator()` 返回 `true`（即元素可被 4 整除）时，`find_if` 返回一个指向该元素的迭代器。然后，将 `find_if` 操作的结果与容器的 `end()` 进行比较，以核实是否找到了能被 4 整除的元素，如第 30 行所示。接下来使用迭代器 `iElement` 显示该元素的值，如第 33 行所示。

一元谓词被大量用于 STL 算法中。例如，`std::partition` 算法使用一元谓词来划分范围，`stable_partition` 算法也使用一元谓词来划分范围，但保持元素的相对顺序不变。诸如 `std::find_if` 等查找函数以及 `std::remove_if` 等删除元素的函数也使用一元谓词，其中 `std::remove_if` 删除指定范围内满足谓词条件的元素。

### 22.2.3 二元函数

如果函数 `f(x, y)` 根据输入参数返回一个值，它将很有用。这种二元函数可用于对两个操作数执行



运算，如加、减、乘、除等。下面的二元函数返回输入参数的积：

```
template <typename elementType>
class CMultiply
{
public:
    elementType operator () (const elementType& elem1,
                             const elementType& elem2)
    {
        return (elem1 * elem2);
    }
};
```

同样，在上述实现中最重要的是 operator()，它接受两个参数并返回它们的积。在 std::transform 等算法中，可使用该二元函数计算两个容器内容的乘积。程序清单 22.5 演示了如何在 std::transform 中使用该二元函数。

#### 程序清单 22.5 使用二元函数将两个范围相乘

```
1: #include <vector>
2: #include <iostream>
3: #include <algorithm>
4:
5: template <typename elementType>
6: class CMultiply
7: {
8: public:
9:     elementType operator () (const elementType& elem1,
10:                             const elementType& elem2)
11:     {
12:         return (elem1 * elem2);
13:     }
14: };
15:
16: int main ()
17: {
18:     using namespace std;
19:
20:     // Create two sample vector of integers with 10 elements each
21:     vector <int> vecMultiplicand, vecMultiplier;
22:
23:     // Insert sample values 0 to 9
24:     for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
25:         vecMultiplicand.push_back (nCount1);
26:
27:     // Insert sample values 100 to 109
28:     for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
29:         vecMultiplier.push_back (nCount2);
30:
31:     // A third container that holds the result of multiplication
32:     vector <int> vecResult;
33:
34:     // Make space for the result of the multiplication
35:     vecResult.resize (10);
36:
37:     transform (    vecMultiplicand.begin (), // range of multiplicands
38:                   vecMultiplicand.end (), // end of range
39:                   vecMultiplier.begin (), // multiplier values
40:                   vecResult.begin (), // range that holds result
41:                   CMultiply <int> () ); // the function that multiplies
42:
43:     cout << "The contents of the first vector are: " << endl;
44:     for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size (); ++ nIndex1)
45:         cout << vecMultiplicand [nIndex1] << ' ';
46:     cout << endl;
47:
48:     cout << "The contents of the second vector are: " << endl;
49:     for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size (); ++nIndex2)
50:         cout << vecMultiplier [nIndex2] << ' ';
51:     cout << endl << endl;
52:
53:     cout << "The result of the multiplication is: " << endl;
```

```

54:     for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
55:         cout << vecResult [nIndex] << ' ';
56:
57:     return 0;
58: }

```

#### ▼ 输出:

```

The contents of the first vector are:
0 1 2 3 4 5 6 7 8 9
The contents of the second vector are:
100 101 102 103 104 105 106 107 108 109

```

```

The result of the multiplication held in the third vector is:
0 101 204 309 416 525 636 749 864 981

```

#### ▼ 分析:

第5~14行包含类CMultiply，这与前一个代码示例相同。在这个示例中，使用算法std::transform将两个范围的内容相乘，并将结果存储在第三个范围中。在这里，这三个范围分别存储在类型为std::vector的vecMultiplicand、vecMultiplier和vecResult中。在第37~41行，使用std::transform将vecMultiplicand中的每个元素与vecMultiplier中对应的元素相乘，并将结果存储在vecResult中。乘法运算是通过调用二元函数CMultiple::operator()执行的，对源向量和目标向量的每一元素都调用了该函数。operator()的返回值保存在vecResult中。

这个示例演示了如何使用二元函数对STL容器中的元素执行算术运算。

### 22.2.4 二元谓词

接受两个参数并返回一个布尔值的函数是二元谓词。这种函数用于诸如std::sort等STL函数中，程序清单22.6演示了如何使用二元谓词对存储std::string值的容器进行不区分大小写的排序。

程序清单 22.6 对字符串进行不区分大小写排序的二元谓词

```

1: #include <algorithm>
2: #include <string>
3: using namespace std;
4:
5: class CCompareStringNoCase
6: {
7: public:
8:     bool operator () (const string& str1, const string& str2) const
9:     {
10:         string str1LowerCase;
11:         // Assign space
12:         str1LowerCase.resize (str1.size ());
13:         // Convert every character to the lower case
14:         transform ( str1.begin (), str1.end ()
15:                     , str1LowerCase.begin (), tolower );
16:
17:         string str2LowerCase;
18:         str2LowerCase.resize (str2.size ());
19:
20:         transform ( str2.begin (), str2.end ()
21:                     , str2LowerCase.begin (), tolower);
22:
23:         return (str1LowerCase < str2LowerCase);
24:     }
25: };

```

#### ▼ 分析:

在operator()中实现的二元谓词中，第14行和第20行首先使用std::transform将输入字符串转换为小写，然后使用字符串的比较运算符<进行比较。该二元谓词可用于算法std::sort，还可用作提供给关

联容器（如 `std::set`）的谓词参数，如程序清单 22.7 所示。

程序清单 22.7 通过使用二元谓词在 `std::set` 存储一系列名字

```
1: #include <set>
2: #include <iostream>
3:
4: // Insert CCompareStringNoCase from listing 22.6 here
5:
6: int main ()
7: {
8:     typedef set <string, CCompareStringNoCase> SET_NAMES;
9:
10:    // Define a set of string to hold names
11:    SET_NAMES setNames;
12:
13:    // Insert some sample names in to the set
14:    setNames.insert ("Tina");
15:    setNames.insert ("jim");
16:    setNames.insert ("Jack");
17:    setNames.insert ("Sam");
18:
19:    cout << "The sample names in the set are: " << endl;
20:
21:    // Display the names in the set
22:    SET_NAMES::const_iterator iNameLocator;
23:    for ( iNameLocator = setNames.begin ()
24:          ; iNameLocator != setNames.end ()
25:          ; ++ iNameLocator )
26:        cout << *iNameLocator << endl;
27:
28:    cout << "Enter a name you wish to search the set for: ";
29:    string strUserInput;
30:    cin >> strUserInput;
31:
32:    SET_NAMES::iterator iNameFound = setNames.find (strUserInput);
33:
34:    if (iNameFound != setNames.end ())
35:        cout << "'" << *iNameFound << "' was found in the set" << endl;
36:    else
37:        cout << "Name '" << strUserInput << "' was not found in the set";
38:
39:    return 0;
40: }
```

#### ▼ 输出:

```
The sample names in the set are:
Jack
jim
Sam
Tina
Enter a name you wish to search the set for: Jim
'jim' was found in the set
```

#### ▼ 分析:

第 8 行的 `typedef` 声明旨在简化代码，以突出二元谓词 `CCompareStringNoCase` 的所有重要用法，该二元谓词帮助 `std::set` 对其内容进行不区分大小写的排序。第 14~17 行将一些值插入到 `setNames` 中。第 30 行提示用户输入一个要在 `setNames` 中进行搜索的名字。第 32 行使用用户输入的字符串 `Jim` 调用 `find`，该字符串的大小写与集合中的名字 `jim` 的大小写不同。由于谓词 `CCompareStringNoCase`，集合的 `find` 函数能够找到该元素。如果没有这个谓词（读者可以试一试），`std::set` 将调用 `std::string` 实现的运算符 `<`。该运算符进行区分大小写的排序，如果用户输入的名字大小写不匹配，将找不到名字。

很多 STL 算法都使用二元谓词，如删除相邻重复元素的 `std::unique`、排序算法 `std::sort`、排序并保持相对顺序的 `std::stable_sort` 以及对两个范围进行操作的 `std::transform`。这些 STL 算法都需要使用二元谓词。

## 22.3 总结

本章介绍了函数对象。在结构或类中实现函数对象时，它将比简单函数有用得多，因为它也可用于存储与状态相关的信息。本章还介绍了谓词，它是一类特殊的函数对象。另外，还通过一些实际示例说明了谓词的用途。

## 22.4 问与答

问：谓词是一种特殊的函数对象，其特殊之处何在？

答：谓词总是返回布尔值。

问：调用诸如 `remove_if` 等函数时，应使用哪种函数对象？

答：应使用通过构造函数将值作为初始状态的一元谓词。

问：对于 `map` 应使用哪种函数对象？

答：应使用二元谓词。

问：没有返回值的简单函数是否可用作谓词？

答：可以。没有返回值的函数也很有用，例如，可用于显示输入的数据。

## 22.5 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 22.5.1 测验

1. 返回布尔值的一元函数称为什么？
2. 不修改数据也不返回布尔值的函数对象有什么用？请通过示例阐述您的观点。
3. 函数对象这一术语的定义是什么？

### 22.5.2 练习

1. 编写一个一元函数，它可供 `std::for_each` 用来显示输入参数的两倍。
2. 进一步扩展上述谓词，使其能够记录它被调用的次数。
3. 编写一个用于降序排序的二元谓词。



## 第 23 章

# STL 算法

标准模板库 (STL) 中很重要的一部分是通用函数, 这些函数位于头文件 <algorithm> 中, 可帮助操作容器的内容。本章介绍如何使用各种 STL 算法以及如何根据应用程序的需求定制这些算法。

### 23.1 什么是 STL 算法

查找、搜索、删除和计数是一些通用算法, 其应用范围很广。STL 通过通用的模板函数提供了这些算法以及其他的很多算法, 可通过迭代器对容器进行操作。要使用 STL 算法, 程序员必须包含头文件 <algorithm>。

#### 注意

虽然大多数算法都通过迭代器对容器进行操作, 但并非所有算法都对容器进行操作, 因此并非所有算法都需要迭代器。有些算法接受一对值, 例如, swap 将这对值交换。同样, min 和 max 也对值进行操作。

### 23.2 STL 算法的分类

STL 算法分两大类: 非变序算法与变序算法。

#### 23.2.1 非变序算法

不改变容器中元素的顺序和内容的算法称为非变序算法。表 23.1 列出了一些主要的非变序算法。

表 23.1 非变序算法

算法	描述
计数算法	
count	在指定范围内查找值与指定值匹配的所有元素
count_if	在指定范围内查找值满足指定条件的所有元素
搜索算法	
search	在目标范围内, 根据元素相等性 (即运算符 ==) 或指定二元谓词搜索第一个满足条件的序列
search_n	在目标范围内搜索与指定值相等或满足指定谓词的 n 个元素
find	在给定范围内搜索与指定值匹配的的第一个元素
find_if	在给定范围内搜索满足指定条件的第一个元素
find_end	在指定范围内搜索最后一个满足特定条件的序列
find_first_of	在目标范围内搜索指定序列中的任何一个元素第一次出现的位置; 在另一个重载版本中, 它搜索满足指定条件的第一个元素
adjacent_find	在集合中搜索两个相等或满足指定条件的元素
比较算法	
equal	比较两个元素是否相等或使用指定的二元谓词判断两者是否相等
mismatch	使用指定的二元谓词找出两个元素范围的第一个不同的地方
lexicographical_compare	比较两个序列中的元素, 以判断哪个序列更小

23.2.2 变序算法

变序算法改变其操作的序列的元素顺序或内容,表 23.2 列出了 STL 提供的一些最有用的变序算法。

表 23.2 变序算法

算法	描述
初始化算法	
fill	将指定值分配给指定范围中的每个元素
fill_n	将指定值分配给指定范围中的前 n 个元素
generate	将指定函数对象的返回值分配给指定范围中的每个元素
generate_n	将指定函数的返回值分配给指定范围中的前 n 个元素
修改算法	
for_each	对指定范围内的每个元素执行指定的操作。当指定的参数修改了范围时, for_each 将是变序算法
transform	对指定范围中的每个元素执行指定的一元函数
复制算法	
copy	将一个范围复制到另一个范围
copy_backward	将一个范围复制到另一个范围,但在目标范围中将元素的排列顺序反转
删除算法	
remove	将指定范围中包含指定值的元素删除
remove_if	将指定范围中满足指定一元谓词的元素删除
remove_copy	将源范围中除包含指定值外的所有元素复制到目标范围
remove_copy_if	将源范围中除满足指定一元谓词外的所有元素复制到目标范围
unique	比较指定范围内的相邻元素,并删除重复的元素。该算法还有一个重载版本,它使用二元谓词来判断要删除哪些元素
unique_copy	将源范围内的所有元素复制到目标范围,但相邻的重复元素除外
替换算法	
replace	用一个值来替换指定范围中与指定值匹配的所有元素
replace_if	用一个值来替换指定范围中满足指定条件的所有元素
排序算法	
sort	使用指定的排序标准对指定范围内的元素进行排序,排序标准由二元谓词提供。排序可能改变相等元素的相对顺序
stable_sort	类似于 sort,但在排序时保持相对顺序
partial_sort	将源范围内指定数量的元素排序
partial_sort_copy	将源范围内的元素复制到目标范围,同时对它们排序
分区算法	
partition	在指定范围中,将元素分为两组:满足指定一元谓词的元素放在第一个组中,其他元素放在第二组中。不一定会保持集合中元素的相对顺序
stable_partition	与 partition 一样将指定范围分为两组,但保持元素的相对顺序不变
可用于排序容器的算法	
binary_search	用于判断一个元素是否存在于一个排序集合中
lower_bound	根据元素的值或二元谓词判断元素可能插入到排序集合中的第一个位置,并返回一个指向该位置的迭代器
upper_bound	根据元素的值或二元谓词判断元素可能插入到排序集合中的最后一个位置,并返回一个指向该位置的迭代器



## 23.3 STL 算法的应用

要学习表 23.1 和表 23.2 所示 STL 算法的用法，最佳的方法是通过示例。为此，读者可通过代码学习如何使用算法，并在自己的应用程序使用这些算法。

### 23.3.1 计算元素个数与查找元素

算法 `std::count`、`count_if`、`find` 和 `find_if` 可用于对指定范围进行元素计数和查找。程序清单 23.1 所示的代码演示了如何使用这些函数。

程序清单 23.1 对集合使用 `std::count`、`count_if`、`find` 和 `find_if`

```
1: #include <algorithm>
2: #include <vector>
3: #include <iostream>
4:
5: // A unary predicate for the *_if functions
6: template <typename elementType>
7: bool IsEven (const elementType& number)
8: {
9:     // return true if the number is even
10:    return ((number % 2) == 0);
11: }
12:
13: int main ()
14: {
15:     using namespace std;
16:
17:     // A sample container - vector of integers
18:     vector <int> vecIntegers;
19:
20:     // Inserting sample values
21:     for (int nNum = -9; nNum < 10; ++ nNum)
22:         vecIntegers.push_back (nNum);
23:
24:     // Display all elements in the collection
25:     cout << "Elements in our sample collection are: " << endl;
26:     vector <int>::const_iterator iElementLocator;
27:     for ( iElementLocator = vecIntegers.begin ()
28:         ; iElementLocator != vecIntegers.end ()
29:         ; ++ iElementLocator )
30:         cout << *iElementLocator << ' ';
31:
32:     cout << endl << endl;
33:
34:     // Determine the total number of elements
35:     cout << "The collection contains ";
36:     cout << vecIntegers.size () << " elements" << endl;
37:
38:     // Use the count_if algorithm with the unary predicate IsEven:
39:     size_t nNumEvenElements = count_if (vecIntegers.begin (),
40:                                         vecIntegers.end (), IsEven <int> );
41:
42:     cout << "Number of even elements: " << nNumEvenElements << endl;
43:     cout << "Number of odd elements: ";
44:     cout << vecIntegers.size () - nNumEvenElements << endl;
45:
46:     // Use count to determine the number of '0's in the vector
47:     size_t nNumZeroes = count (vecIntegers.begin (), vecIntegers.end (), 0);
48:     cout << "Number of instances of '0': " << nNumZeroes << endl << endl;
49:
50:     cout << "Searching for an element of value 3 using find: " << endl;
51:
52:     // Find a sample integer '3' in the vector using the 'find' algorithm
53:     vector <int>::iterator iElementFound;
```

```

54:     iElementFound = find ( vecIntegers.begin ()    // Start of range
55:                           , vecIntegers.end ()    // End of range
56:                           , 3 );                  // Element to find
57:
58:     // Check if find succeeded
59:     if ( iElementFound != vecIntegers.end () )
60:         cout << "Result: Element found!" << endl << endl;
61:     else
62:         cout << "Result: Element was not found in the collection." << endl;
63:
64:     cout << "Finding the first even number using find_if: " << endl;
65:
66:     // Find the first even number in the collection
67:     vector<int>::iterator iEvenNumber;
68:     iEvenNumber = find_if ( vecIntegers.begin () // Start of range
69:                           , vecIntegers.end ()  // End of range
70:                           , IsEven<int> );      // Unary Predicate
71:
72:     if ( iEvenNumber != vecIntegers.end () )
73:     {
74:         cout << "Number '" << *iEvenNumber << "' found at position [";
75:         cout << distance (vecIntegers.begin (), iEvenNumber);
76:         cout << "]" << endl;
77:     }
78:
79:     return 0;
80: }

```

#### ▼ 输出:

Elements in our sample collection are:  
-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9

The collection contains '19' elements  
Number of even elements: 9  
Number of odd elements: 10  
Number of instances of '0': 1

Searching the collection for an element of value 3 using find:  
Result: Element found!

Finding the first even number in the collection using find\_if:  
Number '-8' found at position [1]

#### ▼ 分析:

函数 count 和 find 不需要谓词, 但 count\_if 和 find\_if 需要。这个谓词是由模板函数 IsEven 实现的, 当输入值为偶数时, 该函数返回 true, 如第 7~11 行所示。这个例子演示了如何将这些 STL 算法用于包含整型对象的 vector 容器 vecIntegers。第 39 行使用 count\_if 计算向量中包含多少个偶数。第 47 行使用 count 计算 vecIntegers 包含多少个零。第 54 行使用算法 find 在向量中查找第一个值为 3 的元素, 而 find\_if 使用谓词 IsEven 查找第一个值为偶数的元素。

可以看到, 谓词 IsEven 可用于对范围执行两种截然不同的操作: 一种是计算满足谓词实现的条件元素个数, 另一种是查找满足谓词的元素。因此, 这个例子还表明, 谓词和 STL 算法可提高代码的可复用性, 使代码更高效且更易于维护。

### 23.3.2 在集合中搜索元素或序列

前面的示例演示了如何在容器中查找元素, 但有时需要查找序列。在这种情况下, 不能使用逐元素查找的查找算法, 而应使用查找序列的 search 或 search\_n, 如程序清单 23.2 所示。

程序清单 23.2 使用 search 和 search\_n 在集合中查找序列

```

1: #include <algorithm>
2: #include <vector>

```

```

3: #include <list>
4: #include <iostream>
5:
6: int main ()
7: {
8:     using namespace std;
9:
10:    // A sample container - vector of integers
11:    vector<int> vecIntegers;
12:
13:    for (int nNum = -9; nNum < 10; ++ nNum)
14:        vecIntegers.push_back (nNum);
15:
16:    // Insert some more sample values into the vector
17:    vecIntegers.push_back (9);
18:    vecIntegers.push_back (9);
19:
20:    // Another sample container - a list of integers
21:    list<int> listIntegers;
22:
23:    for (int nNum = -4; nNum < 5; ++ nNum)
24:        listIntegers.push_back (nNum);
25:
26:    // Display the contents of the collections...
27:    cout << "The contents of the sample vector are: " << endl;
28:    vector<int>::const_iterator iVecElementLocator;
29:    for ( iVecElementLocator = vecIntegers.begin ()
30:         ; iVecElementLocator != vecIntegers.end ()
31:         ; ++ iVecElementLocator )
32:        cout << *iVecElementLocator << ' ';
33:
34:    cout << endl << "The contents of the sample list are: " << endl;
35:    list<int>::const_iterator ilistElementLocator;
36:    for ( ilistElementLocator = listIntegers.begin ()
37:         ; ilistElementLocator != listIntegers.end ()
38:         ; ++ ilistElementLocator )
39:        cout << *ilistElementLocator << ' ';
40:
41:    cout << endl << endl;
42:    cout << "'search' the contents of the list in the vector: " << endl;
43:
44:    // Search the vector for the elements present in the list
45:    vector<int>::iterator iRangeLocated;
46:    iRangeLocated = search ( vecIntegers.begin () // Start of range
47:                          , vecIntegers.end ()   // End of range to search in
48:                          , listIntegers.begin () // Start of range to search for
49:                          , listIntegers.end () ); // End of range to search for
50:
51:    // Check if search found a match
52:    if (iRangeLocated != vecIntegers.end ())
53:    {
54:        cout << "The sequence in the list found a match in the vector at ";
55:        cout << "position: ";
56:        cout << distance (vecIntegers.begin (), iRangeLocated);
57:
58:        cout << endl << endl;
59:    }
60:
61:    cout << "Searching for {9, 9, 9} in the vector using 'search_n': ";
62:    cout << endl;
63:
64:    // Now search the vector for the occurrence of pattern {9, 9, 9}
65:    vector<int>::iterator iPartialRangeLocated;
66:    iPartialRangeLocated = search_n ( vecIntegers.begin () // Start range
67:                                    , vecIntegers.end ()   // End range
68:                                    , 3                     // Count of item to be searched for
69:                                    , 9 );                  // Item to search for
70:
71:    if (iPartialRangeLocated != vecIntegers.end ())
72:    {
73:        cout << "The sequence {9, 9, 9} found a match in the vector at ";
74:        cout << "offset-position: ";
75:        cout << distance (vecIntegers.begin (), iPartialRangeLocated);

```

```

76:
77:     cout << endl;
78: }
79:
80:     return 0;
81: }

```

#### ▼ 输出:

```

The contents of the sample vector are:
-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 9 9
The contents of the sample list are:
-4 -3 -2 -1 0 1 2 3 4

'search' the contents of the list in the vector:
The sequence in the list found a match in the vector at position: 5

Searching for {9, 9, 9} in the vector using 'search_n':
The sequence {9, 9, 9} found a match in the vector at offset-position: 18

```

#### ▼ 分析:

在这个例子中，首先定义了两个容器：一个 vector 和一个 list，并使用一些值对其进行了初始化。第 46 行使用 search 在 vector 中搜索 list，提供给 search 的参数是要在其中进行搜索的范围的开头和结尾，还有要搜索的值在 list 中的范围的开头和结尾。由于这里要在整个 vector 中搜索整个 list 的内容，因此使用它们的成员方法 begin() 和 end() 返回的范围。这说明了迭代器和算法之间的联系非紧密，提供这些迭代器的容器的物理特性对算法没有意义。第 66 行使用 search\_n 搜索序列 {9,9,9} 在 vector 中第一次出现的位置。

### 23.3.3 将容器中的元素初始化为指定值

STL 算法 fill 和 fill\_n 用于将容器的内容设置为指定值。fill 将指定范围内的元素设置为指定值，而 fill\_n 接受的参数包括开始位置、计数 n 以及要设置的值，如程序清单 23.3 所示。

程序清单 23.3 使用 fill 和 fill\_n 设置容器中元素的初始值

```

1: #include <algorithm>
2: #include <vector>
3: #include <iostream>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     // Initialize a sample vector with 3 elements
10:    vector<int> vecIntegers (3);
11:
12:    // fill all elements in the container with value 9
13:    fill (vecIntegers.begin (), vecIntegers.end (), 9);
14:
15:    // Increase the size of the vector to hold 6 elements
16:    vecIntegers.resize (6);
17:
18:    // Fill the three elements starting at offset position 3 with value -9
19:    fill_n (vecIntegers.begin () + 3, 3, -9);
20:
21:    cout << "Contents of the vector are: " << endl;
22:    for (size_t nIndex = 0; nIndex < vecIntegers.size (); ++ nIndex)
23:    {
24:        cout << "Element [" << nIndex << "] = ";
25:        cout << vecIntegers [nIndex] << endl;
26:    }
27:
28:    return 0;
29: }

```

**▼ 输出:**

```
Contents of the vector are:
Element [0] = 9
Element [1] = 9
Element [2] = 9
Element [3] = -9
Element [4] = -9
Element [5] = -9
```

**▼ 分析:**

程序清单 23.3 使用函数 `fill` 和 `fill_n` 将容器的内容初始化为两组不同的值, 如第 13 行和第 19 行所示。注意, 在使用值填充范围前调用了函数 `resize()`, 它实际上创建了随后要填充的元素。`fill` 算法是对整个范围进行操作, 而 `fill_n` 可对范围的一部分进行操作。

`fill` 函数将集合的元素设置为程序员指定的值, 而 `generate` 和 `generate_n` 等 STL 算法可用于将集合初始化为文件的内容或随机值, 如程序清单 23.4 所示。

**程序清单 23.4 使用 `generate` 和 `generate_n` 将集合初始化为随机值**

```
1: #include <algorithm>
2: #include <vector>
3: #include <list>
4: #include <iostream>
5:
6: int main ()
7: {
8:     using namespace std;
9:
10:    vector<int> vecIntegers (10);
11:    generate ( vecIntegers.begin (), vecIntegers.end ()    // range
12:             , rand );    // generator function to be called
13:
14:    cout << "Elements in the vector of size " << vecIntegers.size ();
15:    cout << " assigned by 'generate' are: " << endl << "{";
16:    for (size_t nCount = 0; nCount < vecIntegers.size (); ++ nCount)
17:        cout << vecIntegers [nCount] << " ";
18:
19:    cout << "}" << endl << endl;
20:
21:    list<int> listIntegers (10);
22:    generate_n (listIntegers.begin (), 5, rand);
23:
24:    cout << "Elements in the list of size: " << listIntegers.size ();
25:    cout << " assigned by 'generate_n' are: " << endl << "{";
26:    list<int>::const_iterator iElementLocator;
27:    for ( iElementLocator = listIntegers.begin ()
28:          ; iElementLocator != listIntegers.end ()
29:          ; ++ iElementLocator )
30:        cout << *iElementLocator << ' ';
31:
32:    cout << "}" << endl;
33:
34:    return 0;
35: }
```

**▼ 输出:**

```
Elements in the vector of size 10 assigned by 'generate' are:
{41 18467 6334 26500 19169 15724 11478 29358 26982 24464 }
```

```
Elements in the list of size: 10 assigned by 'generate_n' are:
{5705 28145 23281 16827 9961 0 0 0 0 0 }
```

**▼ 分析:**

程序清单 23.4 使用 `generate` 函数将 `vector` 中的所有元素都填充为由 `rand` 函数提供的随机值。注意, `generate` 函数接受一个范围作为输入, 并对该范围内的每个元素调用指定的函数对象 `rand`; 而 `generate_n`

接受起始位置，调用指定的函数对象 rand count 次，以设置容器中 count 个元素的值。容器中不在指定范围内的元素不受影响。

### 23.3.4 用 for\_each 处理范围内的元素

for\_each 算法对指定范围内的每个元素执行指定的一元函数对象，其用法如下：

```
unaryFunctionObjectType mReturn = for_each ( start_of_range
                                             , end_of_range
                                             , unaryFunctionObject );
```

返回值表明，for\_each 返回用于对指定范围内的每个元素进行处理的函数对象（functor）。这意味着使用结构或类作为函数对象可存储状态信息，并在 for\_each 执行完毕后查询这些信息，如程序清单 23.5 所示。该程序清单使用函数对象显示一个范围内的元素，并使用它计算显示了多少个元素。

程序清单 23.5 使用 for\_each 显示集合的内容

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <string>
5:
6: using namespace std;
7:
8: // Unary function object type invoked by for_each
9: template <typename elementType>
10: class DisplayElementKeepCount
11: {
12: private:
13:     int m_nCount;
14:
15: public:
16:     DisplayElementKeepCount ()
17:     {
18:         m_nCount = 0;
19:     }
20:
21:     void operator () (const elementType& element)
22:     {
23:         ++ m_nCount;
24:         cout << element << ' ';
25:     }
26:
27:     int GetCount ()
28:     {
29:         return m_nCount;
30:     }
31: };
32:
33: int main ()
34: {
35:     vector <int> vecIntegers;
36:
37:     for (int nCount = 0; nCount < 10; ++ nCount)
38:         vecIntegers.push_back (nCount);
39:
40:     cout << "Displaying the vector of integers: " << endl;
41:
42:     // Display the array of integers
43:     DisplayElementKeepCount<int> mIntResult =
44:         for_each ( vecIntegers.begin () // Start of range
45:                  , vecIntegers.end ()   // End of range
46:                  , DisplayElementKeepCount<int> () );// Functor
47:
48:     cout << endl;
49:
50:     // Use the state stored in the return value of for_each!
51:     cout << "" << mIntResult.GetCount () << " elements ";
52:     cout << " in the vector were displayed!" << endl << endl;
```



```

53:
54:     string strSample ("for_each and strings!");
55:     cout << "String sample is: " << strSample << endl << endl;
56:
57:     cout << "String displayed using DisplayElementKeepCount:" << endl;
58:     DisplayElementKeepCount<char> mCharResult = for_each (strSample.begin()
59:                                                         , strSample.end ()
60:                                                         , DisplayElementKeepCount<char> () );
61:
62:     cout << endl;
63:     cout << "' ' << mCharResult.GetCount () << " characters were displayed";
64:
65:     return 0;
66: }

```

### ▼ 输出:

```

Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9
'10' elements in the vector were displayed!

String sample is: for_each and strings!

String displayed using DisplayElementKeepCount:
for_each and strings!
'21' characters were displayed

```

### ▼ 分析:

上述代码不仅演示了如何使用 `for_each`，还说明它返回函数对象这一特点。返回的函数对象可存储信息，如被调用的次数。上述代码定义了两个范围，一个包含在整型 `vector` `vecIntegers` 中，另一个是 `std::string` 对象 `strSample`。第 44 行和第 58 行分别对这两个范围调用了 `for_each`，并将 `DisplayElementKeepCount` 作为函数对象。`for_each` 对指定范围内的每个元素调用 `operator()`，该函数将元素显示在屏幕上，并将内部计数加 1。`for_each` 执行完毕后返回该函数对象，然后调用成员函数 `GetCount()` 告诉用户该函数对象被调用的次数。

将信息（或状态）存储在算法返回的对象中，对实际编程很有帮助。

## 23.3.5 使用 `std::transform` 对范围进行变换

`for_each` 和 `std::transform` 很类似，它们都对源范围内的每个元素调用指定函数对象。然而，`std::transform` 有两个版本：一个接受一元函数，另一个接受二元函数，因此 `transform` 算法还可以处理一对来自两个不同范围的元素。

`transform` 函数的两个版本都将指定变换函数的结果赋给指定的目标范围，这与 `for_each` 不同，`for_each` 只处理一个范围。程序清单 23.6 演示了 `std::transform` 的用法。

### 程序清单 23.6 使用一元函数和二元函数的 `std::transform`

```

1: #include <algorithm>
2: #include <string>
3: #include <vector>
4: #include <deque>
5: #include <iostream>
6: #include <functional>
7:
8: int main ()
9: {
10:     using namespace std;
11:
12:     string strSample ("THIS is a TEst string!");
13:     cout << "The sample string is: " << strSample << endl;
14:

```

```

15:     string strLowerCaseCopy;
16:     strLowerCaseCopy.resize (strSample.size ());
17:
18:     transform ( strSample.begin ()           // start of source range
19:                , strSample.end ()           // end of source range
20:                , strLowerCaseCopy.begin ()   // start of destination range
21:                , tolower );                 // unary function
22:
23:     cout << "Result of 'transform' on the string with 'tolower':" << endl;
24:     cout << "\"" << strLowerCaseCopy << "\"" << endl << endl;
25:
26:     // Two sample vectors of integers...
27:     vector <int> vecIntegers1, vecIntegers2;
28:     for (int nNum = 0; nNum < 10; ++ nNum)
29:     {
30:         vecIntegers1.push_back (nNum);
31:         vecIntegers2.push_back (10 - nNum);
32:     }
33:
34:     // A destination range for holding the result of addition
35:     deque <int> dqResultAddition (vecIntegers1.size ());
36:
37:     transform ( vecIntegers1.begin ()        // start of source range 1
38:                , vecIntegers1.end ()        // end of source range 1
39:                , vecIntegers2.begin ()      // start of source range 2
40:                , dqResultAddition.begin ()  // start of destination range
41:                , plus <int> () );           // binary function
42:
43:     cout << "Result of 'transform' using binary function 'plus': " << endl;
44:     cout << endl << "Index   Vector1 + Vector2 = Result (in Deque)" << endl;
45:     for (size_t nIndex = 0; nIndex < vecIntegers1.size (); ++ nIndex)
46:     {
47:         cout << nIndex << "      \t " << vecIntegers1 [nIndex] << "\t+   ";
48:         cout << vecIntegers2 [nIndex] << " \t =   ";
49:
50:         cout << dqResultAddition [nIndex] << endl;
51:     }
52:
53:     return 0;
54: }

```

### ▼ 输出:

The sample string is: THIS is a TEst string!  
 Result of using 'transform' with unary function 'tolower' on the string:  
 "this is a test string!"

Result of 'transform' using binary function 'plus':

Index	Vector1	+	Vector2	=	Result (in Deque)
0	0	+	10	=	10
1	1	+	9	=	10
2	2	+	8	=	10
3	3	+	7	=	10
4	4	+	6	=	10
5	5	+	5	=	10
6	6	+	4	=	10
7	7	+	3	=	10
8	8	+	2	=	10
9	9	+	1	=	10

### ▼ 分析:

该示例演示了 `std::transform` 的两种版本的用法：一种版本使用一元函数 `tolower` 处理一个范围，另一个版本使用二元函数 `plus` 处理两个范围内。第一个版本如第 18~21 行所示，它修改字符串的大小写，将每个字符都改为小写。如果使用 `toupper` 而不是 `tolower`，将把字符变为大写。`std::transform` 的另一个版本如第 37~41 行所示，它对来自两个输入范围（这里是两个 `vector`）内的元素进行操作：使用一个二元谓词——STL 函数 `plus`（由头文件 `<functional>` 提供）将两个元素相加。`std::transform` 每次处理一对数，它将这对数提供给二元函数 `plus`，然后将结果赋给目标范围中的元素——这里是 `std::deque` 中

的元素。注意，这里使用另一种容器来存储结果只是出于演示目的。这个例子表明，通过使用迭代器，可将容器及其实现同 STL 算法分离；transform 是一种处理范围的算法，它无需知道实现这些范围的容器的细节。因此，虽然这里的输入范围为 vector，而输出范围为 deque，但该算法仍管用——只要指定范围的边界（提供给 transform 的输入参数）有效。

### 23.3.6 复制和删除操作

STL 提供了两个重要的复制函数：copy 和 copy\_backward。copy 沿向前的方向将源范围的内容赋给目标范围，而 copy\_backward 沿向后的方向将源范围的内容赋给目标范围。

remove 将容器中与指定值匹配的元素删除，而 remove\_if 使用一个一元谓词并将容器中满足该谓词的元素删除。

程序清单 23.7 演示了这些复制和删除函数的用法。

程序清单 23.7 copy、copy\_backward、remove 和 remove\_if 的用法

```
1: #include <algorithm>
2: #include <vector>
3: #include <list>
4: #include <iostream>
5:
6: // A unary predicate for the remove_if function
7: template <typename elementType>
8: bool IsOdd (const elementType& number)
9: {
10:     // returns true if the number is odd
11:     return ((number % 2) == 1);
12: }
13:
14: int main ()
15: {
16:     using namespace std;
17:
18:     // A list with sample values
19:     list <int> listIntegers;
20:
21:     for (int nCount = 0; nCount < 10; ++ nCount)
22:         listIntegers.push_back (nCount);
23:
24:     cout << "Elements in the source (list) are: " << endl;
25:
26:     // Display all elements in the collection
27:     list <int>::const_iterator iElementLocator;
28:     for ( iElementLocator = listIntegers.begin ()
29:           ; iElementLocator != listIntegers.end ()
30:           ; ++ iElementLocator )
31:         cout << *iElementLocator << ' ';
32:
33:     cout << endl << endl;
34:
35:     // Initialize the vector to hold twice as many elements as the list
36:     vector <int> vecIntegers (listIntegers.size () * 2);
37:
38:     vector <int>::iterator iLastPos;
39:     iLastPos = copy ( listIntegers.begin () // start of source range
40:                     , listIntegers.end ()   // end of source range
41:                     , vecIntegers.begin () ); // start of destination range
42:
43:     // Now, use copy_backward to copy the same list into the vector
44:     copy_backward ( listIntegers.begin ()
45:                   , listIntegers.end ()
46:                   , vecIntegers.end () );
47:
48:     cout << "Elements in the destination (vector) after copy: " << endl;
49:
50:     // Display all elements in the collection
```

```

51:     vector<int>::const_iterator iDestElementLocator;
52:     for ( iDestElementLocator = vecIntegers.begin ()
53:           ; iDestElementLocator != vecIntegers.end ()
54:           ; ++ iDestElementLocator )
55:         cout << *iDestElementLocator << ' ';
56:
57:     cout << endl << endl;
58:
59:     /*
60:      Remove all instances of '0':
61:      std::remove does not change the size of the container,
62:      it simply moves elements forward to fill gaps created
63:      and returns the new 'end' position.
64:     */
65:     vector<int>::iterator iNewEnd;
66:     iNewEnd = remove (vecIntegers.begin (), vecIntegers.end (), 0);
67:
68:     // Use this new 'end position' to resize vector
69:     vecIntegers.erase (iNewEnd, vecIntegers.end ());
70:
71:     // Remove all odd numbers from the vector using remove_if
72:     iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
73:                          IsOdd<int>);    // The predicate
74:
75:     vecIntegers.erase (iNewEnd , vecIntegers.end ());
76:
77:     cout << "Elements in the destination (vector) after remove: " << endl;
78:
79:     // Display all elements in the collection
80:     for ( iDestElementLocator = vecIntegers.begin ()
81:           ; iDestElementLocator != vecIntegers.end ()
82:           ; ++ iDestElementLocator )
83:         cout << *iDestElementLocator << ' ';
84:
85:     return 0;
86: }

```

#### ▼ 输出:

Elements in the source (list) are:  
0 1 2 3 4 5 6 7 8 9

Elements in the destination (vector) after copy:  
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

Elements in the destination (vector) after remove:  
2 4 6 8 2 4 6 8

#### ▼ 分析:

该示例首先定义了函数 `IsOdd`。顾名思义，当输入的数字为奇数时，该函数返回 `true`。该谓词供 STL 算法 `remove_if` 使用，如第 72 行所示。`copy` 和 `remove` 函数都返回指向末尾位置的迭代器。`copy` 函数返回的迭代器可用于从该位置开始向前复制更多元素；函数 `remove` 或 `remove_if` 返回的迭代器可在后续的 `erase` 操作中用于纠正容器的大小，如代码所示。`remove` 或 `remove_if` 在删除元素时，将该元素后面的元素向前移一个位置，即覆盖要删除的元素。然而，它不会改变容器的大小或减少报告的元素数，除非程序员使用它返回的迭代器（这里为 `iNewEnd`）执行 `erase` 操作。第 66 行调用 `remove` 将集合中所有为零的元素删除，紧接着执行 `vector::erase` 操作以纠正容器的大小。第 39 行演示了如何使用 `copy` 将整型 `list` 的内容复制到 `vector` 中。注意，`copy` 函数只接受目标范围的起始位置，它假定目标范围足够大，能够存储 `copy` 操作的结果。这就是将 `vecIntegers` 初始化为至少能够存储源 `list` `listIntegers` 中所有元素的原因。如第 44 行所示，`copy_backward` 也进行复制，但复制的方向相反。注意，这个函数使用的第三个参数是目标范围的末尾。

如果指定的是目标容器的末尾，则 `copy_backward` 与下述 `copy` 等效：

```

copy ( listIntegers.begin () // start of source range
      , listIntegers.end ()   // end of source range
      , iLastPos );           // returned by previous 'copy'

```

### 23.3.7 替换值以及替换满足给定条件的元素

STL 算法 `replace` 与 `replace_if` 分别用于替换集合中等于指定值和满足给定条件的元素。前者根据比较运算符`==`的返回值来替换元素，而后者需要一个用户指定的一元谓词，对于要替换的每个值，该谓词都返回 `true`。程序清单 23.8 演示了这两个函数的用法。

程序清单 23.8 使用 `replace` 和 `replace_if` 在指定范围内替换值

```
1: #include <iostream>
2: #include <algorithm>
3: #include <vector>
4:
5: // The unary predicate used by replace_if to replace even numbers
6: bool IsEven (const int & nNum)
7: {
8:     return ((nNum % 2) == 0);
9: }
10:
11: int main ()
12: {
13:     using namespace std;
14:
15:     // Initialize a sample vector with 6 elements
16:     vector<int> vecIntegers (6);
17:
18:     // fill first 3 elements with value 8
19:     fill (vecIntegers.begin (), vecIntegers.begin () + 3, 8);
20:
21:     // fill last 3 elements with value 5
22:     fill_n (vecIntegers.begin () + 3, 3, 5);
23:
24:     // shuffle the container
25:     random_shuffle (vecIntegers.begin (), vecIntegers.end ());
26:
27:     cout << "The initial contents of the vector are: " << endl;
28:     for (size_t nIndex = 0; nIndex < vecIntegers.size (); ++ nIndex)
29:     {
30:         cout << "Element [" << nIndex << "] = ";
31:         cout << vecIntegers [nIndex] << endl;
32:     }
33:
34:     cout << endl << "Using 'std::replace' to replace value 5 by 8" << endl;
35:     replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
36:
37:     cout << "Using 'std::replace_if' to replace even values by -1" << endl;
38:     replace_if (vecIntegers.begin (), vecIntegers.end (), IsEven, -1);
39:
40:     cout << endl << "Contents of the vector after replacements:" << endl;
41:     for (size_t nIndex = 0; nIndex < vecIntegers.size (); ++ nIndex)
42:     {
43:         cout << "Element [" << nIndex << "] = ";
44:         cout << vecIntegers [nIndex] << endl;
45:     }
46:
47:     return 0;
48: }
```

#### ▼ 输出:

```
The initial contents of the vector are:
Element [0] = 5
Element [1] = 8
Element [2] = 5
Element [3] = 8
Element [4] = 8
Element [5] = 5
```

```
Using 'std::replace' to replace value 5 by 8
```

Using 'std::replace\_if' to replace even values by -1

Contents of the vector after replacements:

```
Element [0] = -1
Element [1] = -1
Element [2] = -1
Element [3] = -1
Element [4] = -1
Element [5] = -1
```

### ▼ 分析:

该示例给整型 vector `vecIntegers` 填充值, 然后使用 STL 算法 `std::random_shuffle` 将这些值打乱, 如第 25 行。第 35 行使用了 `replace` 将所有的 5 替换为 8。因此, 第 38 行使用 `replace_if` 将所有偶数替换为 -1 后, 集合包含 6 个元素, 每个元素的值都为 -1, 如输出所示。

## 23.3.8 排序、在有序集合中搜索以及删除重复元素

在实际的应用程序中, 经常需要排序以及在有序范围内 (出于性能的考虑) 进行搜索。经常需要对一组信息进行排序, 而有时需要根据用户的要求对有序的信息进行过滤。同样, 在显示集合的内容前, 需要删除重复的元素。程序清单 23.9 使用 STL 算法 `std::sort` 将一个范围排序, 使用 `std::binary_search` 在有序的范围进行搜索, 然后使用 `std::unique` 删除相邻的重复元素 (排序后相同的元素将成为邻居)。

程序清单 23.9 使用 `std::sort`、`binary_search` 和 `unique`

```
1: #include <algorithm>
2: #include <vector>
3: #include <string>
4: #include <iostream>
5:
6: int main ()
7: {
8:     using namespace std;
9:     typedef vector <string> VECTOR_STRINGS;
10:
11:     // A vector of strings
12:     VECTOR_STRINGS vecNames;
13:
14:     // Insert sample values
15:     vecNames.push_back ("John Doe");
16:     vecNames.push_back ("Jack Nicholson");
17:     vecNames.push_back ("Sean Penn");
18:     vecNames.push_back ("Anna Hoover");
19:
20:     // insert a duplicate into the vector
21:     vecNames.push_back ("Jack Nicholson");
22:
23:     cout << "The initial contents of the vector are:" << endl;
24:     for (size_t nItem = 0; nItem < vecNames.size (); ++ nItem)
25:     {
26:         cout << "Name [" << nItem << "] = \"";
27:         cout << vecNames [nItem] << "\" << endl;
28:     }
29:
30:     cout << endl;
31:
32:     // sort the names using std::sort
33:     sort (vecNames.begin (), vecNames.end ());
34:
35:     cout << "The sorted vector contains names in the order:" << endl;
36:     for (size_t nItem = 0; nItem < vecNames.size (); ++ nItem)
37:     {
38:         cout << "Name [" << nItem << "] = \"";
39:         cout << vecNames [nItem] << "\" << endl;
40:     }
41:
42:     cout << endl;
```



```
43:
44:     cout << "Searching for \"John Doe\" using 'binary_search':" << endl;
45:     bool bElementFound = binary_search (vecNames.begin (), vecNames.end (),
46:                                         "John Doe");
47:
48:     // Check if search found a match
49:     if (bElementFound)
50:         cout << "Result: \"John Doe\" was found in the vector!" << endl;
51:     else
52:         cout << "Element not found " << endl;
53:
54:     cout << endl;
55:
56:     VECTOR_STRINGS::iterator iNewEnd;
57:
58:     // Erase adjacent duplicates
59:     iNewEnd = unique (vecNames.begin (), vecNames.end ());
60:     vecNames.erase (iNewEnd, vecNames.end ());
61:
62:     cout << "The contents of the vector after using 'unique':" << endl;
63:     for (size_t nItem = 0; nItem < vecNames.size (); ++ nItem)
64:     {
65:         cout << "Name [" << nItem << "] = \"";
66:         cout << vecNames [nItem] << "\" << endl;
67:     }
68:
69:     return 0;
70: }
```

#### ▼ 输出:

The initial contents of the vector are:

```
Name [0] = "John Doe"
Name [1] = "Jack Nicholson"
Name [2] = "Sean Penn"
Name [3] = "Anna Hoover"
Name [4] = "Jack Nicholson"
```

The sorted vector contains names in the order:

```
Name [0] = "Anna Hoover"
Name [1] = "Jack Nicholson"
Name [2] = "Jack Nicholson"
Name [3] = "John Doe"
Name [4] = "Sean Penn"
```

Searching for "John Doe" using 'binary\_search':

Result: "John Doe" was found in the vector!

The contents of the vector after using 'unique':

```
Name [0] = "Anna Hoover"
Name [1] = "Jack Nicholson"
Name [2] = "John Doe"
Name [3] = "Sean Penn"
```

#### ▼ 分析:

`binary_search` 等搜索算法只能用于有序容器, 对未排序的 `vector` 使用该算法可能得到意外结果。因此, 上述代码首先将 `vector` `vecNames` 排序, 如第 33 行所示, 然后使用 `binary_search` 在该 `vector` 中查找 John Doe。同样, 第 59 行和第 60 行使用 `std::unique` 删除第二个相邻的重复元素。在大多数情况下, 该算法也只适用于有序集合。

#### 注意

`stable_sort` 的用法与 `sort` 类似。`stable_sort` 确保排序后元素的相对顺序保持不变。为确保相对顺序保持不变, 将降低性能, 这是一个需要考虑的因素, 尤其在元素的相对顺序不重要时。

### 23.3.9 将范围分区

`std::partition` 用于将输入范围分为两部分: 一部分满足一元谓词, 另一个不满足。然而, `std::partition`

不保证每个分区中元素的相对顺序不变。要保持相对顺序不变, 应使用 `std::stable_partition`。程序清单 23.10 演示了这两个算法的用法。

程序清单 23.10 使用 `partition` 和 `stable_partition` 将整型范围分为偶数值和奇数值

---

```

1: #include <algorithm>
2: #include <vector>
3: #include <iostream>
4:
5: bool IsEven (const int& nNumber)
6: {
7:     return ((nNumber % 2) == 0);
8: }
9:
10: int main ()
11: {
12:     using namespace std;
13:
14:     // a sample collection...
15:     vector<int> vecIntegers;
16:
17:     // fill sample values 0 - 9, in that order
18:     for (int nNum = 0; nNum < 10; ++ nNum)
19:         vecIntegers.push_back (nNum);
20:
21:     // a copy of the sample vector
22:     vector<int> vecCopy (vecIntegers);
23:
24:     // separate even values from the odd ones - even comes first.
25:     partition (vecIntegers.begin (), vecIntegers.end (), IsEven);
26:
27:     // display contents
28:     cout << "The contents of the vector after using 'partition' are:";
29:     cout << endl << "{";
30:
31:     for (size_t nItem = 0; nItem < vecIntegers.size (); ++ nItem)
32:         cout << vecIntegers [nItem] << ' ';
33:
34:     cout << "}" << endl << endl;
35:
36:     // now use stable_partition on the vecCopy - maintains relative order
37:     stable_partition (vecCopy.begin (), vecCopy.end (), IsEven);
38:
39:     // display contents of vecCopy
40:     cout << "The effect of using 'stable_partition' is: " << endl << "{";
41:
42:     for (size_t nItem = 0; nItem < vecCopy.size (); ++ nItem)
43:         cout << vecCopy [nItem] << ' ';
44:
45:     cout << "}" << endl << endl;
46:
47:     return 0;
48: }

```

---

#### ▼ 输出:

```

The contents of the vector after using 'partition' are:
{0 8 2 6 4 5 3 7 1 9 }

```

```

The effect of using 'stable_partition' is:
{0 2 4 6 8 1 3 5 7 9 }

```

#### ▼ 分析:

上述代码将包含在 `vector vecIntegers` 的整型范围分为偶数和奇数。第一次分区是在第 25 行使用 `std::partition` 完成的, 第二次是在第 37 行使用 `stable_partition` 完成的。为便于比较, 这里将范围 `vecIntegers` 复制到 `vecCopy` 中, 并对前者使用 `std::partition`, 对后者使用 `std::stable_partition`。与使用 `partition` 相比, 使用 `stable_partition` 的效果很明显, 如输出所示。 `stable_partition` 保持每个分区中元素的相对顺序不变。

注意，保持顺序是有代价的，这种代价可能很小（如在这个例子中），也可能很大，这取决于包含在范围内的对象类型。`stable_partition` 的速度比 `partition` 慢，因此只应在容器中元素的相对顺序很重要时才使用它。

### 23.3.10 在有序集合中插入元素

将元素插入到有序集合中时，经常需要将其插入到正确位置。为满足这种需求，STL 提供了 `lower_bound` 和 `upper_bound` 等函数。

`lower_bound` 和 `upper_bound` 分别返回在不破坏现有顺序的情况下，元素可插入到有序范围内的最前位置和最后位置。

程序清单 23.11 演示了如何使用 `lower_bound` 将元素插入到在有序的人名 `list` 中的最前位置。

程序清单 23.11 对有序范围使用 `lower_bound` 和 `upper_bound`

```
1: #include <algorithm>
2: #include <list>
3: #include <string>
4: #include <iostream>
5:
6: int main ()
7: {
8:     using namespace std;
9:
10:    typedef list <string> LIST_STRINGS;
11:
12:    // A sample list of strings
13:    LIST_STRINGS listNames;
14:
15:    // Insert sample values
16:    listNames.push_back ("John Doe");
17:    listNames.push_back ("Brad Pitt");
18:    listNames.push_back ("Jack Nicholson");
19:    listNames.push_back ("Sean Penn");
20:    listNames.push_back ("Anna Hoover");
21:
22:    // Sort all the names in the list
23:    listNames.sort ();
24:
25:    cout << "The sorted contents of the list are: " << endl;
26:    LIST_STRINGS::iterator iNameLocator;
27:    for ( iNameLocator = listNames.begin ()
28:          ; iNameLocator != listNames.end ()
29:          ; ++ iNameLocator )
30:    {
31:        cout << "Name [" << distance (listNames.begin (), iNameLocator);
32:        cout << "] = \"" << *iNameLocator << "\" << endl;
33:    }
34:
35:    cout << endl;
36:
37:    LIST_STRINGS::iterator iMinInsertPosition;
38:
39:    // The closest / lowest position where the element can be inserted
40:    iMinInsertPosition = lower_bound ( listNames.begin (), listNames.end ()
41:                                     , "Brad Pitt" );
42:
43:    LIST_STRINGS::iterator iMaxInsertPosition;
44:
45:    // The farthest / highest position where an element may be inserted
46:    iMaxInsertPosition = upper_bound ( listNames.begin (), listNames.end ()
47:                                     , "Brad Pitt" );
48:
49:    cout << "The lowest index where \"Brad Pitt\" can be inserted is: ";
50:    cout << distance (listNames.begin (), iMinInsertPosition) << endl;
51:
52:    cout << "The highest index where \"Brad Pitt\" can be inserted is: ";
```

```
53:     cout << distance (listNames.begin (), iMaxInsertPosition) << endl;
54:
55:     cout << endl;
56:
57:     cout << "Inserting \"Brad Pitt\" in the sorted list:" << endl;
58:     listNames.insert (iMinInsertPosition, "Brad Pitt");
59:
60:     cout << "The contents of the list now are: " << endl;
61:     for ( iNameLocator = listNames.begin ()
62:           ; iNameLocator != listNames.end ()
63:           ; ++ iNameLocator )
64:     {
65:         cout << "Name [" << distance (listNames.begin (), iNameLocator);
66:         cout << "] = \"" << *iNameLocator << "\" << endl;
67:     }
68:
69:     return 0;
70: }
```

#### ▼ 输出:

```
The sorted contents of the list are:
Name [0] = "Anna Hoover"
Name [1] = "Brad Pitt"
Name [2] = "Jack Nicholson"
Name [3] = "John Doe"
Name [4] = "Sean Penn"

The lowest index where "Brad Pitt" can be inserted is: 1
The highest index where "Brad Pitt" can be inserted is: 2

Inserting "Brad Pitt" in the sorted list:
The contents of the list now are:
Name [0] = "Anna Hoover"
Name [1] = "Brad Pitt"
Name [2] = "Brad Pitt"
Name [3] = "Jack Nicholson"
Name [4] = "John Doe"
Name [5] = "Sean Penn"
```

#### ▼ 分析:

可将元素插入到有序集合的两个位置：一个是 `lower_bound` 返回的最前位置（离集合开头最近），另一个是 `upper_bound` 返回的迭代器，它是最后位置（离集合开头最远）。在程序清单 23.11 中，要插入到有序集合的字符串 Brad Pitt 已包含在集合中，因此最前位置和最后位置不同（否则，两者将相同）。在第 40 行和第 46 行分别调用了这两个函数。如输出所示，第 58 行使用 `lower_bound` 返回的迭代器将字符串插入 list 后，list 仍处于有序状态。也就是说，可将元素插入到集合的某个位置而不破坏集合内容的有序状态。使用 `upper_bound` 返回的迭代器也如此。

## 23.4 总结

本章介绍了 STL 中最重要、功能最强大的方面：算法。在本章中，读者了解了各种类型的算法，并通过示例对其用法有了更清晰的认识。

## 23.5 问与答

问：诸如 `std::transform` 等变序算法能否用于关联容器（如 `std::set`）？

答：即使可以，也不应这样做。应将关联容器的内容视为常量，这是因为关联容器在插入元素时进行排序，因此元素的相对位置不仅对 `find` 等函数很重要，对容器的效率也很重要。因此，不应将诸如 `std::transform` 等变序算法用于 STL `set`。

问：要将顺序容器的每个元素都设置为特定的值，可使用 `std::transform` 吗？

答：虽然可以使用 `std::transform`，但使用 `fill` 或 `fill_n` 更合适。

问：`copy_backward` 是否会反转目标容器中元素的排列顺序？

答：不会。STL 算法 `copy_backward` 按相反的顺序复制元素，但不改变元素的排列顺序，即它从范围末尾开始复制到开头。如果要反转集合中元素的排列顺序，应使用 `std::reverse`。

问：是否应对 `list` 使用 `std::sort`？

答：`std::sort` 可用于 `list`，用法与用于其他顺序容器一样。然而，`list` 需要保持一个特殊特征：对 `list` 的操作不会导致现有迭代器失效，而 `std::sort` 不能保证该特征得以保持。因此，STL `list` 通过成员函数 `list::sort` 提供了 `sort` 算法。应使用该函数，因为它确保指向 `list` 中元素的迭代器不会失效，即使元素的相对位置发生了变化。

问：为什么在将元素插入到有序范围中时，使用 `lower_bound` 或 `upper_bound` 等函数很重要？

答：这两个函数分别提供有序集合中的第一个位置和最后一个位置，将元素插入这些位置时不会破坏集合的有序状态。

## 23.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 23.6.1 测验

1. 要将 `list` 中满足特定条件的元素删除，应使用 `std::remove_if` 还是 `list::remove_if`？
2. 假设有一个包含 `CContactItem` 对象的 `list`，在没有显式指定二元谓词时，`list::sort` 函数将如何对这些元素进行排序？
3. STL 算法 `generate` 将调用 `generator` 函数多少次？
4. `std::transform` 与 `std::for_each` 之间的区别何在？

### 23.6.2 练习

1. 编写一个二元谓词，它接受字符串作为输入参数，并根据不区分大小写的比较结果返回一个值。
2. 演示 STL 算法（如 `copy`）如何使用迭代器实现其功能——复制两个类型不同的容器存储的序列，而无需知道目标集合的特征。
3. 您正在编写一个应用程序，它按星星在地平线上升起的顺序记录它们的特点。在天文学中，星球的大小很重要，其升起和落下的相对顺序亦如此。如果要根据星星的大小对这个集合进行排序，应使用 `std::sort` 还是 `std::stable_sort`？

# 第 24 章

## 自适应容器：栈和队列

标准模板库 (STL) 提供了一些这样的容器，即使用其他容器模拟栈和队列的行为。这种内部使用一种容器但呈现其他容器的行为特征的容器称为自适应容器 (adaptive container)。

在本章中，您将学习：

- 栈和队列的行为特征
- 使用 STL stack
- 使用 STL queue
- 使用 STL priority\_queue

### 24.1 栈和队列的行为特征

栈和队列与数组或列表极其相似，但对插入、访问和删除元素的方式有一定的限制。可将元素插入到什么位置以及可从什么位置删除元素决定了容器的行为特征。

#### 24.1.1 栈

栈是 LIFO (后进先出) 系统，只能从栈顶插入或删除元素。可将栈视为一叠盘子，最后叠上去的盘子将首先被取下来，而不能取下中间或底部的盘子。

泛型 STL 容器 `std::stack` 模拟了栈的这种行为；要使用 `std::stack`，必须包含头文件 `<stack>`。

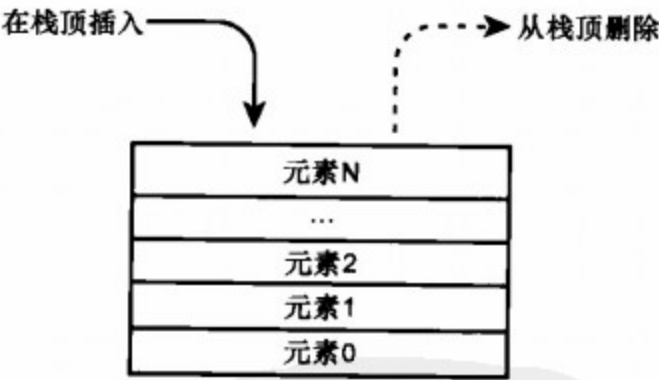


图 24.1 对栈的操作

#### 24.1.2 队列

队列是 FIFO (先进先出) 系统，元素被插入到队尾，最先插入的元素最先删除。可将队列视为一系列在邮局排队购买邮票的人：先加入队列的人先离开。

泛型 STL 容器 `std::queue` 模拟了队列的这种行为。要使用 `std::queue`，必须包含头文件 `<queue>`。

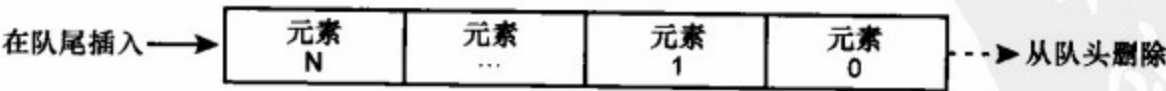


图 24.2 队列的操作

### 24.2 使用 STL stack 类

STL stack 是一个模板类，要使用它，必须包含头文件 `<stack>`。它是一个泛型类，允许在顶部插入和删除元素，而不允许访问中间的元素。从这种角度看，`std::stack` 的行为很像一叠盘子。



### 24.2.1 实例化 stack

在有些 STL 实现中，std::stack 的定义如下：

```
template <
    class elementType,
    class Container=deque<Type>
> class stack;
```

参数 elementType 是 stack 存储的对象的类型。第二个模板参数 Container 是 stack 使用的默认底层容器实现类。stack 默认在内部使用 std::deque 来存储数据。实例化模板类 stack 时，通过显式地设置第二个模板参数，可指定在内部使用 vector 或 list 来存储数据。程序清单 24.1 演示了如何实例化模板类 std::stack。

程序清单 24.1 实例化 STL stack

```
1: #include <stack>
2: #include <vector>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // A stack of integers
9:     stack <int> stackIntegers;
10:
11:    // A stack of doubles
12:    stack <double> stackDoubles;
13:
14:    // A stack of doubles contained in a vector
15:    stack <double, vector <double> > stackDoublesInVector;
16:
17:    return 0;
18: }
```

▼ 分析：

该示例没有输出，但演示了如何实例化 STL 模板 stack。第 9 行和第 11 行实例化了两个 stack 对象，分别用于存储类型为 int 和 double 的元素。第 15 行也实例化了一个用于存储 double 元素的 stack，但将第二个模板参数（stack 在内部使用的集合类）指定为 vector。如果没有指定第二个模板参数，stack 将自动使用默认的 std::deque。

### 24.2.2 stack 的成员函数

stack 改变了其他容器（如 deque、list 或 vector）的行为，通过限制元素插入或删除的方式实现其功能，从而提供严格遵守栈机制的行为特征。表 24.1 解释了 stack 类的公有成员函数并演示了如何将

表 24.1 std::stack 的成员函数	
函数	描述
push	在栈顶插入元素 stackIntegers.push (25);
pop	删除栈顶的元素 stackIntegers.pop ();
empty	检查栈是否为空并返回一个布尔值 if (stackIntegers.empty ()) DoSomething ();
size	返回栈中的元素数 size_t nNumElements = stackIntegers.size ();
top	获得指向栈顶元素的引用 cout << "Element at the top = " << stackIntegers.top ();

如表所示，stack 的公有成员函数只提供了这样的方法，即插入或删除元素的位置符合栈的行为特征。也就是说，虽然底层容器可能是 deque、vector 或 list，但禁用了这些容器的有些功能，以实现的行为特征。

程序清单 24.2 演示了如何在 stack 中插入和删除元素。

程序清单 24.2 使用整型 stack

---

```

1: #include <stack>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // A stack of integers
9:     stack <int> stackIntegers;
10:
11:     // Push sample values to the top of the stack
12:     cout << "Pushing numbers {25, 10, -1, 5} into the stack:" << endl;
13:
14:     // push = insert at top of the container
15:     stackIntegers.push (25);
16:     stackIntegers.push (10);
17:     stackIntegers.push (-1);
18:     stackIntegers.push (5);
19:     // So, 25 is at the bottom and 5 is at the top!
20:
21:     cout << "The stack contains " << stackIntegers.size () << " elements";
22:     cout << endl;
23:
24:     // pop = remove the topmost element
25:     cout << endl << "Popping them one after another..." << endl;
26:
27:     while (stackIntegers.size () != 0)
28:     {
29:         cout << "The element at the top is: " << stackIntegers.top();
30:         cout << endl << "Removing it from the stack!" << endl;
31:
32:         // Remove the topmost element
33:         stackIntegers.pop ();
34:     }
35:
36:     if (stackIntegers.empty ())
37:         cout << endl << "The stack is now empty!";
38:
39:     return 0;
40: }
```

---

#### ▼ 输出:

```

Pushing numbers {25, 10, -1, 5} into the stack:
The stack contains 4 elements
```

```

Popping them one after another...
The element at the top is: 5
Removing it from the stack!
The element at the top is: -1
Removing it from the stack!
The element at the top is: 10
Removing it from the stack!
The element at the top is: 25
Removing it from the stack!
```

```

The stack is now empty!
```

#### ▼ 分析:

上述示例首先使用函数 `stack::push` 将一些值插入到整型 stack `stackIntegers` 中，然后将元素从 stack 中删除。stack 只允许访问最栈顶元素，可使用成员 `stack::pop()` 访问栈顶元素，如第 29 行所示。使用 `stack::pop()` 可每次从 stack 中删除一个元素，如第 33 行所示。第 33 行所属的 while 循环确保不断执行 `pop()` 操作，直到 stack 为

空。从元素弹出的顺序可知，最后插入的元素最先弹出，这说明了 stack 的典型 LIFO（后进先出）特征。

程序清单 24.2 演示了 stack 的所有 5 个成员函数。注意，stack 类用作底层容器的所有 STL 顺序容器都提供了 push\_back 和 insert，但它们不是 stack 的公有成员函数；用于访问非容器顶部元素的迭代器也如此。stack 只暴露了栈顶元素，而没有暴露其他任何元素。

## 24.3 使用 STL queue 类

STL queue 是一个模板类，要使用它，必须包含头文件 <queue>。queue 是一个泛型类，只允许在末尾插入元素以及从开头删除元素。queue 不允许访问中间的元素，但可以访问开头和末尾的元素。从这种意义上说，std::queue 的行为与超市收银台前的队列极其相似。

### 24.3.1 实例化 queue

std::queue 的定义如下：

std::queue is defined as

```
template <
    class elementType,
    class Container = deque<Type>
> class queue;
```

其中 elementType 是 queue 对象包含的元素的类型。Container 是 std::queue 用于存储其数据的集合类型，可将该默认参数设置为 std::list、vector 或 deque，默认为 deque。

程序清单 24.3 演示了如何实例化模板 std::queue。

程序清单 24.3 实例化 STL queue

---

```
1: #include <queue>
2: #include <list>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // A queue of integers
9:     queue <int> qIntegers;
10:
11:    // A queue of doubles
12:    queue <double> qDoubles;
13:
14:    // A queue of doubles stored internally in a list
15:    queue <double, list <double> > qDoublesInList;
16:
17:    return 0;
18: }
```

---

#### ▼ 分析：

上述示例演示了如何实例化 STL 泛型类 queue，第 9 行创建一个整型 queue，而第 12 行创建了一个双精度型 queue。第 15 行实例化 queue qDoublesInList 时，显式地指定 queue 使用底层容器 std::list 来管理内部数据，这是通过第二个模板参数指定的。如果指定第二个模板参数（就像实例化前两个 queue 那样），默认将使用底层容器 std::deque 来管理 queue 的内容。

### 24.3.2 queue 的成员函数

与 std::stack 一样，std::queue 的实现也是基于 STL 容器 vector、list 或 deque 的。Queue 提供了几个成员函数来实现队列的行为特征。表 24.2 通过程序清单 24.3 所示的整型 queue qIntegers 解释了 queue 的成员函数。

表 24.2

std::queue 的成员函数

函数	描述
push	在队尾（即最后一个位置）插入一个元素 qIntegers.push (25);
pop	将队首（即最开始位置）的元素删除 qIntegers.pop ();
front	返回指向队首元素的引用 cout << "Element at front: " << qIntegers.front ();
back	返回指向队尾元素（即最后插入的元素）的引用 cout << "Element at back: " << qIntegers.back ();
empty	检查队列是否为空并返回一个布尔值 if (qIntegers.empty ()) cout << "The queue is empty!";
size	返回队列中的元素数 size_t nNumElements = qIntegers.size ();

STL queue 没有提供 begin()和 end()等函数，而大多数 STL 容器都提供了这些函数，包括 queue 类在底层使用的 deque、vector 或 list。这是有意为之的，旨在只允许对 queue 执行符合队列行为特征的操作。注意，与 stack 不同，queue 允许查看其两端的元素，因此提供了函数 front 和 back。插入只能在队尾进行，删除只能在队首进行，程序清单 24.4 演示了这种行为。

程序清单 24.4 使用整型 queue

```
1: #include <queue>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // A queue of integers
9:     queue <int> qIntegers;
10:
11:     cout << "Inserting {10, 5, -1, 20} into the queue" << endl;
12:
13:     // elements pushed into the queue are inserted at the end
14:     qIntegers.push (10);
15:     qIntegers.push (5);
16:     qIntegers.push (-1);
17:     qIntegers.push (20);
18:     // the elements in the queue now are {20, -1, 5, 10} in that order
19:
20:     cout << "The queue contains " << qIntegers.size ();
21:     cout << " elements" << endl;
22:     cout << "Element at the front: " << qIntegers.front() << endl;
23:     cout << "Element at the back: " << qIntegers.back ();
24:     cout << endl << endl;
25:
26:     cout << "Removing them one after another..." << endl;
27:     while (qIntegers.size () != 0)
28:     {
29:         cout << "Deleting element " << qIntegers.front () << endl;
30:
31:         // Remove the element at the front of the queue
32:         qIntegers.pop ();
33:     }
34:
35:     cout << endl;
36:
37:     // Test if the queue is empty
38:     if (qIntegers.empty ())
39:         cout << "The queue is now empty!";
40:
41:     return 0;
42: }
```

**▼ 输出:**

```
Inserting {10, 5, -1, 20} into the queue
The queue contains 4 elements
Element at the front: 10
Element at the back: 20
```

```
Removing them one after another...
```

```
Deleting element 10
Deleting element 5
Deleting element -1
Deleting element 20
```

```
The queue is now empty!
```

**▼ 分析:**

在第 14~17 行, 使用 `push` 在队列 `queueIntegers` 的末尾插入元素。第 22 行和第 23 行分别使用函数 `front()` 和 `back()` 引用队首和队尾的元素。第 27~33 行的 `while` 循环显示队首的元素, 然后使用 `pop()` 删除它, 直到队列为空。从输出可知, 元素被删除的顺序与插入顺序相同。

## 24.4 使用 STL 优先级队列

STL `priority_queue` 是一个模板类, 要使用它, 也必须包含头文件 `<queue>`。`priority_queue` 与 `queue` 的不同之处在于, 包含最大值 (或二元谓词认为是最大值) 的元素位于队首, 且只能在队首执行操作。

### 24.4.1 实例化 `priority_queue` 类

`std::priority_queue` 类的定义如下:

```
template <
    class element_type,
    class Container=vector<Type>,
    class Compare=less<typename Container::value_type>
>
class priority_queue
```

其中 `element_type` 是一个模板参数, 指定了优先级队列将包含的元素的类型。第二个模板参数指定 `priority_queue` 在内部将使用哪个集合类来存储数据, 第三个参数让程序员能够指定一个二元谓词, 以帮助队列判断哪个元素应位于队首。如果没有指定二元谓词, `priority_queue` 类将默认使用 `std::less`, 它使用运算符 < 比较对象。

程序清单 24.5 演示了如何实例化 `priority_queue`。

#### 程序清单 24.5 实例化 STL `priority_queue`

```
1: #include <queue>
2:
3: int main ()
4: {
5:     using namespace std;
6:
7:     // A priority queue of integers sorted using std::less <> (default)
8:     priority_queue <int> pqIntegers;
9:
10:    // A priority queue of doubles
11:    priority_queue <double> pqDoubles;
12:
13:    // A priority queue of integers sorted using std::greater <>
14:    priority_queue <int, deque <int>, greater <int> > pqIntegers_Inverse;
15:
16:    return 0;
17: }
```

▼ 分析:

第 8 行和第 11 行实例化了两个 `priority_queue`，其中前者用于存储 `int` 对象，而后者用于存储 `double` 对象。由于没有指定其他模板参数，因此将默认使用 `std::vector` 作为内部数据的容器，并默认使用 `std::less` 提供的比较标准。因此，这两个队列将包含的值最大的元素放在队首。然而，实例化 `pqIntegers_Inverse` 时，通过第二个参数指定使用 `deque` 作为内部容器，并将谓词指定为 `std::greater`，该谓词导致值最小的元素位于队首（程序清单 24.7 说明了使用该谓词的结果）。

24.4.2 `priority_queue` 的成员函数

`queue` 提供了成员函数 `front()` 和 `back()`，但 `priority_queue` 没有。表 24.3 简要地介绍了 `priority_queue` 的成员函数。

表 24.3 `std::priority_queue` 的成员函数

函数	描述
<code>push</code>	在优先级队列中插入一个元素 <code>pqIntegers.push (10);</code>
<code>pop</code>	删除队首元素，即最大的元素 <code>pqIntegers.pop ();</code>
<code>top</code>	返回指向队列中最大元素（即队首元素）的引用 <code>pqIntegers. cout &lt;&lt; "The largest element inpriority queue is: " &lt;&lt; pqIntegers.top ();</code>
<code>empty</code>	检查优先级队列是否为空并返回一个布尔值 <code>if (pqIntegers.empty ())</code> <code>cout &lt;&lt; "The queue is empty!";</code>
<code>size</code>	返回优先级队列中的元素个数 <code>size_t nNumElements = pqIntegers.size ();</code>

从该表可知，只能使用 `top()` 来访问队列的成员，该函数返回值最大的元素，最大的元素是根据用户指定的谓词或默认的 `std::less` 确定的。程序清单 24.6 演示了如何使用 `priority_queue` 的成员函数。

程序清单 24.6 使用 `priority_queue`

```
1: #include <queue>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     priority_queue <int> pqIntegers;
9:     cout << "Inserting {10, 5, -1, 20} into the priority_queue" << endl;
10:
11:     // elements get push-ed into the p-queue
12:     pqIntegers.push (10);
13:     pqIntegers.push (5);
14:     pqIntegers.push (-1);
15:     pqIntegers.push (20);
16:
17:     cout << "The queue contains " << pqIntegers.size () << " elements";
18:     cout << endl;
19:     cout << "Element at the top: " << pqIntegers.top () << endl << endl;
20:
21:     while (!pqIntegers.empty ())
22:     {
23:         cout << "Deleting the topmost element: " << pqIntegers.top ();
24:         cout << endl;
25:
26:         pqIntegers.pop ();
27:     }
28:
29:     return 0;
30: }
```



**▼ 输出:**

```
Inserting {10, 5, -1, 20} into the priority_queue
The queue contains 4 elements
Element at the top: 20
```

```
Deleting the topmost element: 20
Deleting the topmost element: 10
Deleting the topmost element: 5
Deleting the topmost element: -1
```

**▼ 分析:**

与前一个使用 `std::queue()` 的示例（程序清单 24.4）一样，这个示例也将一些整型值插入到 `priority_queue` 中，然后使用 `pop` 函数删除队首元素，如第 26 行所示。从输出可知，值最大的元素位于队首，因此调用 `priority_queue::pop` 将删除容器中值最大的元素。

下一个示例（程序清单 24.7）使用谓词 `std::greater<int>` 实例化一个 `priority_queue`。该谓词导致优先级队列认为包含的数字最小的元素为最大的元素，并将其放在队首。

**程序清单 24.7 通过使用谓词将值最小的元素放在 `priority_queue` 开头**

```
1: #include <queue>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // Define a priority_queue object with greater <int> as predicate
9:     // So, numbers of smaller magnitudes are evaluated as greater in value
10:    priority_queue <int, vector <int>, greater <int> > pqIntegers;
11:
12:    cout << "Inserting {10, 5, -1, 20} into the priority queue" << endl;
13:
14:    // elements get push-ed into the p-queue
15:    pqIntegers.push (10);
16:    pqIntegers.push (5);
17:    pqIntegers.push (-1);
18:    pqIntegers.push (20);
19:
20:    cout << "The queue contains " << pqIntegers.size () << " elements";
21:    cout << endl;
22:    cout << "Element at the top: " << pqIntegers.top () << endl << endl;
23:
24:    while (!pqIntegers.empty ())
25:    {
26:        cout << "Deleting the topmost element " << pqIntegers.top ();
27:        cout << endl;
28:
29:        // delete the number at the 'top'
30:        pqIntegers.pop ();
31:    }
32:
33:    return 0;
34: }
```

**▼ 输出:**

```
Inserting {10, 5, -1, 20} into the priority queue
The queue contains 4 elements
Element at the top: -1
```

```
Deleting the topmost element -1
Deleting the topmost element 5
Deleting the topmost element 10
Deleting the topmost element 20
```

**▼ 分析:**

在这个示例中，大多数代码以及提供给 `priority_queue` 的所有值都与前一个示例（程序清单 24.6）

相同，但输出表明这两个队列的行为不同。这个 `priority_queue` 使用谓词 `greater <int>` 比较其元素，如第 10 行所示。该谓词导致包含的整数最小的元素被认为是最大的，因此放在队首。这样，第 26 行使用的函数 `top()` 总是显示 `priority_queue` 中最小的整数，然后第 30 行使用 `pop()` 将其删除。

因此，弹出元素时，该 `priority_queue` 按升序弹出整数。

## 24.5 总结

本章阐述了 3 个重要的自适应容器——STL `stack`、`queue` 和 `priority_queue`。这些容器使用顺序容器并对其进行改造，以满足其内部数据存储需求，再通过成员函数呈现出栈与队列独特的行为特征。

## 24.6 问与答

问：能否修改栈中间的元素？

答：不能，这不符合栈的行为特征。

问：能否对队列中的所有元素进行迭代？

答：队列不支持迭代器，只能访问队尾的元素。

问：STL 算法能否用于自适应容器？

答：STL 算法使用迭代器。由于 `stack` 和 `queue` 类都没有提供标记范围两端的迭代器，因此无法将 STL 算法用于这些容器。

## 24.7 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 24.7.1 测验

1. 能否修改 `priority_queue` 的行为，使得值最大的元素最后弹出？
2. 假设有一个包含 `Ccoins` 对象的 `priority_queue`，要让 `priority_queue` 将币值最大的硬币放在队首，需要为 `Ccoins` 定义哪种成员运算符？
3. 假设有一个包含 6 个 `CCoins` 对象的 `stack`，能否访问或删除第一个插入的 `CCoins` 对象？

### 24.7.2 练习

1. 邮局有一个包含人（`Cperson` 类）的队列。`Cperson` 包含两个成员属性，分别用于存储年龄和性别，其定义如下：

```
class CPerson
{
public:
    int m_nAge;
    bool m_bIsFemale;
};
```

请编写一个二元谓词，帮助 `priority_queue` 优先向老人和妇女提供服务。

2. 编写一个程序，使用 `stack` 类反转用户输入的字符串的排列顺序。

## 第 25 章

# 使用 STL 位标志

位是存储设置与标志的高效方法。STL 提供了可帮助组织与操作位信息的类。

在本章中，您将学习：

- bitset 类
- vector<bool>

### 25.1 bitset 类

std::bitset 是一个 STL 类，用于处理以位和位标志表示的信息。std::bitset 不是 STL 容器类，因为它不能调整长度，也不具备容器的其他特征，如通过迭代器进行访问。这是一个实用类，针对处理长度在编译阶段已知的位序列进行了优化。

#### 实例化 std::bitset

要使用这个模板类，必须包含头文件<bitset>，并在实例化时通过模板参数指定其实例需要管理的位数。

程序清单 25.1 演示了如何实例化模板类 bitset。

程序清单 25.1 实例化 std::bitset

---

```
1: #include <bitset>
2: #include <iostream>
3: #include <string>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     // instantiate a bitset object for holding 4 bits
10:    // all initialized to '0000'
11:    bitset <4> fourBits;
12:    cout << "The initial contents of fourBits: " << fourBits << endl;
13:
14:    // instantiate a bitset object for holding 5 bits
15:    // initialize it to a bit sequence supplied by a string
16:    bitset <5> fiveBits (string ("10101"));
17:    cout << "The initial contents of fiveBits: " << fiveBits << endl;
18:
19:    // instantiate a bitset object for 8 bits
20:    // given an unsigned long init value
21:    bitset <8> eightbits (255);
22:    cout << "The initial contents of eightBits: " << eightbits << endl;
23:
24:    return 0;
25: }
```

---

▼ 输出:

```
The initial contents of fourBits: 0000
The initial contents of fiveBits: 10101
The initial contents of eightBits: 11111111
```

▼ 分析:

该示例演示了 3 种创建 `bitset` 对象的方法：通过默认构造函数将位序列初始化为 0（如第 11 行所示），通过 STL `string` 以字符串形式指定位序列（如第 16 行所示），通过 `unsigned long` 指定二进制序列对应的十进制值（如第 21 行所示）。注意，在每个实例中，都需要通过一个模板参数指定位序列包含的位数。位数在编译阶段指定，而不是动态的。指定 `bitset` 的位数后，便不能插入更多的位，而不像 `vector` 那样可以调整在编译阶段指定的长度。

25.2 使用 `std::bitset` 及其成员

`bitset` 类提供了很多成员函数，可用于在 `bitset` 中插入位、设置或重置内容、读取内容或将内容写入到流中。它还提供了一些运算符，用于显示位序列以及执行按位逻辑运算。

25.2.1 `std::bitset` 的运算符

第 13 章介绍了运算符，还介绍了运算符最重要的作用是提高类的可用性。`std::bitset` 提供一些很有用的运算符，如表 25.1 所示，这些运算符使得使用 `bitset` 的非常简单。表 25.1 通过程序清单 25.1 所示的 `bitset` 对象 `fourBits` 演示这些运算符的用法。

表 25.1 `std::bitset` 提供的运算符

运算符	描述
运算符<<	将位序列的文本表示插入到输出流中 cout << fourBits;
运算符>>	将一个字符串插入到 <code>bitset</code> 对象中 "0101" >> fourBits;
运算符&	执行按位与操作 bitset <4> result (fourBits1 & fourBits2);
运算符	执行按位或操作 bitwise <4> result (fourBits1   fourBits2);
运算符^	执行按位异或操作 bitwise <4> result (fourBits1 ^ fourBits2);
运算符~	执行按位取反操作 bitwise <4> result (~fourBits1);
运算符 >>=	执行按位右移操作 fourBits >>= (2); //右移两位
运算符<<=	执行按位左移操作 fourBits <<= (2); // 左移两位
运算符[N]	返回指向位序列中第 (N+1) 位的引用 fourBits [2] = 0; // 将第 3 位设置为 0 bool bNum = fourBits [2]; //读取第 3 位

除这些运算符外，`std::bitset` 还提供了 `|=`、`&=`、`^=`和`~=`等运算符，用于对 `bitset` 对象执行按位操作。

25.2.2 `std::bitset` 的成员方法

位可以存储两种状态：要么是已设置（1），要么是重置（0）。要对 `bitset` 的内容进行操作，可使用

表 25.2 列出的成员函数对 bitset 中的一位或所有位进行操作。

函数	描述
set	将序列中的所有位都设置为 1 fourBits.set (); //现在序列包含 1111
set (N, val=1)	将第 N+1 位设置为 val 指定的值 (默认为 1) fourBits.set (2, 0); // 将第 3 位设置为 0
reset	将序列中的所有位都重置为 0 fourBits.reset (); // 现在序列包含 0000
reset (N)	将偏移位置为 (N+1) 的位清除 fourBits.reset (2); //现在第 3 位的值为 0
flip	将位序列中的所有位取反 fourBits.flip (); // 0101 将变为 1010
size	返回序列中的位数 size_t nNumBits = fourBits.size (); // 返回 4
count	返回序列中值为 1 的位数 size_t nNumBitsSet = fourBits.count (); size_t nNumBitsReset = fourBits.size () - fourBits.count ();

程序清单 25.2 演示了成员方法和运算符的用法。

程序清单 25.2 使用 bitset 执行逻辑运算

```
1: #include <bitset>
2: #include <string>
3: #include <iostream>
4:
5: int main ()
6: {
7:     using namespace std;
8:
9:     // A bitset to hold 8-bits
10:    bitset <8> eightBits;
11:    cout << "Enter a 8-bit sequence: ";
12:
13:    // Store user-supplied sequence into the bitset
14:    cin >> eightBits;
15:    cout << endl;
16:
17:    // Supply info on number of 1s and 0s in it:
18:    cout << "The number of 1s in the input sequence: ";
19:    cout << eightBits.count () << endl;
20:    cout << "The number of 0s in the input sequence: ";
21:    cout << eightBits.size () - eightBits.count () << endl;
22:
23:    // create a copy
24:    bitset <8> flipInput (eightBits);
25:
26:    // flip the bits
27:    flipInput.flip ();
28:    cout << "The flipped version of the input sequence is: "
29:        << flipInput << endl << endl;
30:
31:    // another 8-bit sequence to perform bitwise-ops against the first
32:    bitset <8> eightMoreBits;
33:    cout << "Enter another 8-bit sequence: ";
34:    cin >> eightMoreBits;
35:    cout << endl;
36:
37:    cout << "Result of AND, OR and XOR between the two sequences:" << endl;
38:    cout << eightBits << " & " << eightMoreBits << " = "
39:        << (eightBits & eightMoreBits) // bitwise AND
40:        << endl;
```

```

41:
42:     cout << eightBits << " | " << eightMoreBits << " = "
43:           << (eightBits | eightMoreBits)    // bitwise OR
44:           << endl;
45:
46:     cout << eightBits << " ^ " << eightMoreBits << " = "
47:           << (eightBits ^ eightMoreBits)    // bitwise XOR
48:           << endl;
49:
50:     return 0;
51: }

```

### ▼ 输出:

```

Enter a 8-bit sequence: 11100101

The number of 1s in the input sequence: 5
The number of 0s in the input sequence: 3
The flipped version of the input sequence is: 00011010

Enter another 8-bit sequence: 10010111

Result of AND, OR and XOR between the two sequences:
11100101 & 10010111 = 10000101
11100101 | 10010111 = 11110111
11100101 ^ 10010111 = 01110010

```

### ▼ 分析:

该示例是一个交互式程序，它不仅演示了使用 `std::bitset` 在两个位序列之间执行按位运算很简单，还演示了如何使用 `std::bitset` 的流运算符。运算符 `>>` 用于将位序列打印到屏幕上，而运算符 `<<` 用于读取用户以字符串形式输入的位序列。第 14 行将用户提供的序列填充到 `eightBits` 中。第 19 行使用 `count()` 获得序列中值为 1 的位数。序列中值为零的位数是通过将返回 `bitset` 中位数的 `size()` 与 `count()` 相减得到的，如第 21 行所示。`flipInput` 位于 `eightBits` 的一个复本的开头，然后使用 `flip()` 将序列中所有位取反，如第 27 行所示。现在它包含的序列中每位的值都与原来相反。其他代码演示了对两个 `bitset` 执行 AND、OR 和 XOR 等操作的结果。

注意，这个 STL 类有一个缺点，就是它不能动态地调整长度，即仅当在编辑阶段知道序列将存储多少位时才能使用 `bitset`。为克服这种缺点，STL 向程序员提供了 `vector<bool>` 类（在有些 STL 实现中为 `bit_vector`）。

## 25.3 vector<bool>

`vector<bool>` 是对 `std::vector` 的部分具体化，用于存储布尔数据。这个类可动态地调整长度，因此程序员无需在编译阶段知道要存储的布尔标志数。

### 25.3.1 实例化 vector<bool>

要使用 `vector<bool>`，必须包含头文件 `<vector>`。下面的示例（程序清单 25.3）演示了如何实例化 `vector<bool>`。

#### 程序清单 25.3 实例化 vector<bool>

```

1: #include <vector>
2:
3: int main ()
4: {
5:     using namespace std;
6:
7:     // Instantiate an object using the default constructor
8:     vector <bool> vecBool1;

```



```

9:
10: // A vector of 10 elements with value true (default: false)
11: vector<bool> vecBool2 (10, true);
12:
13: // Instantiate one object as a copy of another
14: vector<bool> vecBool2Copy (vecBool2);
15:
16: return 0;
17: }

```

### ▼ 分析:

该示例演示了一些创建 vector<bool>对象的方法：第 8 行使用默认构造函数，第 11 行创建了一个包含 10 个布尔标志的对象，其中每个标志的值都为 true，第 14 行演示了如何通过复制 vector<bool>对象来创建另一个 vector<bool>对象。

## 25.3.2 使用 vector<bool>

vector<bool>提供了函数 flip()，用于将序列中的布尔值取反。除这个方法外，vector<bool>与 std::vector 极其相似，例如，可使用 push\_back 将标志位插入到序列中。下面的示例（程序清单 25.4）更详细地演示了这个类的用法。

### 程序清单 25.4 使用 vector<bool>

```

1: #include <vector>
2: #include <iostream>
3:
4: int main ()
5: {
6:     using namespace std;
7:
8:     // Instantiate a vector<bool> to hold 3 elements
9:     vector<bool> vecBool (3);
10:
11:     // Assign 3 elements using the array operator []
12:     vecBool [0] = true;
13:     vecBool [1] = true;
14:     vecBool [2] = false;
15:
16:     // Insert a 4th element using push_back:
17:     // this will cause the vector to resize the buffer
18:     vecBool.push_back (true);
19:
20:     cout << "The contents of the vector are: " << endl << "{";
21:     for (size_t nIndex = 0; nIndex < vecBool.size (); ++ nIndex)
22:         cout << vecBool [nIndex] << ' ';
23:     cout << "}" << endl << endl;
24:
25:     vecBool.flip ();
26:
27:     cout << "The flipped contents of the vector are: " << endl << "{";
28:     for (size_t nIndex = 0; nIndex < vecBool.size (); ++ nIndex)
29:         cout << vecBool [nIndex] << ' ';
30:     cout << "}" << endl;
31:
32:     return 0;
33: }

```

### ▼ 输出:

```

The contents of the vector are:
{1 1 0 1 }

```

```

The flipped contents of the vector are:
0 0 1 0 }

```

### ▼ 分析:

在这个示例中,通过运算符[]访问了向量中的布尔标志,如第22行所示,这与普通向量极其相似。第25行使用函数 flip()将位标志取反,即将所有0都转换为1,将所有1都转换为0。

## 25.4 总结

本章介绍了用于处理位序列和位标志最有效的工具:std::bitset 类。另外还介绍 vector<bool>类,它也可以存储布尔标志——其位数不需要在编译时就确定。

## 25.5 问与答

问:在可以使用 std::bitset 和 vector<bool>的情况下,您将选择使用哪个类来存储二进制标志?

答:bitset,因为它更适合这种需求。

问:假设有一个名为 myBitSeq 的 std::bitset 对象,它包含一定数量的位。如何确定它包含多少个值为0(或 false)的位?

答:bitset::count()返回值为1的位数,bitset::size()返回总位数,将后者减去前者将得到序列中值为0的位数。

问:能否使用迭代器来访问 vector<bool>中的元素?

答:可以。vector<bool>是 std::vector 的部分具体化,它支持迭代器。

问:能否在编译阶段指定要存储在 vector<bool>中的元素个数?

答:可以。为此,可调用重载构造函数并指定元素数,也可在实例化后调用 vector<bool>::resize 函数。

## 25.6 作业

作业包括测验和练习,前者帮助加深读者对所学知识的理解,后者提供了使用新学知识的机会。请尽量先完成测验和练习题,然后再对照附录D的答案,继续学习下一章前,请务必弄懂这些答案。

### 25.6.1 测验

1. bitset 能否扩展其内部缓冲区以存储可变的元素数?
2. 为什么 bitset 不属于 STL 容器类?
3. 您会使用 std::vector 来存储位数在编译阶段就知道的固定位数吗?

### 25.6.2 练习

1. 创建一个长4位的 bitset 对象,并使用一个数字来初始化它,然后显示结果并将其与另一个 bitset 对象相加(注意:bitsets 不支持语法 bitsetA = bitsetX + bitsetY)。
2. 请演示如何将 bitset 对象中的位取反。



## 第五部分

### 高级 C++ 概念

第 26 章 理解智能指针

第 27 章 处理流

第 28 章 处理异常

第 29 章 杂项内容

## 第 26 章

# 理解智能指针

管理堆（或自由存储区）中的内存时，C++程序员并非一定要使用常规指针，而可使用更智能的指针。在本章中，您将学习：

- 什么是智能指针以及为什么需要智能指针
- 智能指针是如何实现的
- 不同类型的智能指针
- C++标准库提供的智能指针类 `auto_ptr`
- 流行的智能指针库

### 26.1 什么是智能指针

简单地说，C++中的智能指针是包含重载运算符的类，其行为像常规指针，但智能指针能够及时、妥善地销毁动态分配的数据，并实现了定义良好的对象生命周期管理策略，因此更有价值。

#### 26.1.1 使用常规（原始）指针有何问题

与其他现代编程语言不同，C++在内存的分配、释放以及管理方面向程序员提供了全面的灵活性。不幸的是，这种灵活性是把双刃剑，一方面，它使 C++成为一种功能强大的语言，另一方面，它让程序员能够制造与内存相关的问题，如动态分配的对象没有正确地释放时将导致内存泄露。

例如：

```
CData *pData = mObject.GetData ();
/*
    Questions: Is the object pointed to by pData dynamically allocated?
    Who will perform the deallocation: caller or the called, if necessary?
    Answer: No idea!
*/
pData->Display ();
```

在上述代码中，没有显而易见的方法获悉 `pData` 指向的内存：

- 是否是从堆中分配的，因此最终需要释放；
- 是否由调用者负责释放；
- 对象的析构函数是否会自动销毁该对象。

虽然这种不明确性可通过添加注释以及遵循编码实践来部分缓解，但这些机制太松散了，无法有效地避免因滥用动态分配的数据和指针而导致的错误。

#### 26.1.2 智能指针有何帮助

鉴于使用常规指针以及常规的内存管理方法存在的问题，当 C++程序员需要管理堆或自由存储区

中的数据时，并非一定要使用它们，而可在程序中使用智能指针，从而以更智能的方式分配和管理内存：

```
smart_pointer<CData> spData = mObject.GetData ();

// Use a smart pointer like a conventional pointer!
spData->Display ();
(*spData).Display ();

// Don't have to worry about de-allocation
// (the smart pointer's destructor does it for you)
```

智能指针的行为类似常规指针（这里将其称为原始指针），但通过重载的运算符和析构函数确保动态分配的数据能够及时地销毁，从而提供了更多有用的功能。

## 26.2 智能指针是如何实现的

这个问题暂时可以简化为：“智能指针 `spData` 是如何做到像常规指针的？”答案如下：智能指针类重载了运算符`*`（解除引用运算符）和运算符`->`（成员选择运算符），让程序员可以像使用常规指针那样使用它们。运算符重载在第 13 章讨论过。

另外，为让您能够在堆中管理各种类型，几乎所有良好的智能指针类都是模板类，包含其功能的泛型实现。由于是模板，它们是通用的，可以根据要管理的对象类型进行具体化。

程序清单 26.1 是一个简单智能指针类的实现。

程序清单 26.1 智能指针类最基本的组成部分

```
1: template <typename T>
2: class smart_pointer
3: {
4: private:
5:     T* m_pRawPointer;
6: public:
7:     smart_pointer (T* pData) : m_pRawPointer (pData) {} // constructor
8:     ~smart_pointer () {delete pData;}; // destructor
9:
10:    // copy constructor
11:    smart_pointer (const smart_pointer & anotherSP);
12:    // assignment operator
13:    smart_pointer& operator= (const smart_pointer& anotherSP);
14:
15:    T& operator* () const // dereferencing operator
16:    {
17:        return *(m_pRawPointer);
18:    }
19:
20:    T* operator-> () const // member selection operator
21:    {
22:        return m_pRawPointer;
23:    }
24: };
```

### ▼ 分析：

该智能指针类实现了两个运算符`*`和`->`，如第 15~18 行与第 20~24 行所示，它们让这个类能够用作常规意义上的“指针”。例如，如果有一个 `CDog` 类，则可这样对该类型的对象使用智能指针：

```
smart_pointer <CDog> pSmartDog (new CDog);
pSmartDog->Bark ();
int nAge = (*pSmartDog).GetAge ();
```

这个 `smart_pointer` 类还没有实现使其非常智能，从而胜于常规指针的功能。构造函数（如第 7 行所示）接受一个指针，并将其保存到该智能指针类内部的一个指针对象中。析构函数释放该指针，从而实现了自动内存释放。

使智能指针真正“智能”的是复制构造函数、赋值运算符和析构函数的实现，它们决定了智能指针对象在被传递给函数、赋值或离开作用域（即像其他类对象一样被销毁）时的行为。介绍完整的智



能指针实现前，需要了解一些智能指针类型。

## 26.3 智能指针类型

管理内存资源（即实现的内存所有权模型）是智能指针类与众不同的地方。智能指针决定在复制和赋值时如何处理内存资源。最简单的实现通常会导致性能问题，而最快的实现可能并非适合所有应用程序。因此，在应用程序中使用智能指针前，程序员应理解其工作原理。

智能指针的分类实际上就是内存资源管理策略的分类，可分为如下几类：

- 深度复制，
- 写时复制（Copy on Write, COW），
- 引用计数，
- 引用链接，
- 破坏性复制。

下面首先简要地介绍一下这些策略，然后再探索 C++ 标准库提供的智能指针 `std::auto_ptr`。

### 26.3.1 深度复制

在实现深度复制的智能指针中，每个智能指针实例都保存一个它管理的对象的完整副本。每当智能指针被复制时，也将复制它指向的对象（因此称为深度复制）。每当智能指针离开作用域时，将（通过析构函数）释放它指向的内存。

虽然基于深度复制的智能指针看起来不比按值传递对象高，但在处理多态对象时，其优点将显现出来。如下所示，使用智能指针可避免切片（slicing）问题：

```
// Example of Slicing When Passing Polymorphic Objects by Value
// CAnimal is a base class for CDog and CCat.
void MakeAnimalTalk (CAnimal mAnimal)    // note parameter type
{
    mAnimal.Talk (); // virtual function
}

// ... Some function
CCat mCat;
MakeAnimalTalk (mCat);
// Slicing: only the CAnimal part of mCat is sent to MakeAnimalTalk

CDog mDog;
MakeAnimalTalk (mDog);    // Slicing again
```

如果程序员选择使用深度复制智能指针，如程序清单 26.2 所示，便可解决切片问题。

程序清单 26.2 使用基于深度复制的智能指针将多态对象作为其基类对象进行传递

```
1: template <typename T>
2: class deepcopy_smart_pointer
3: {
4: private:
5:     T* m_pObject;
6: public:
7:     //... other functions
8:
9:     // copy constructor of the deepcopy pointer
10:    deepcopy_smart_pointer (const deepcopy_smart_pointer& source)
11:    {
12:        // Use a virtual clone function defined in the derived class
13:        // to get a complete copy of the object
14:        m_pObject = source->Clone ();
15:    }
16: };
17:
```



```
18: void MakeAnimalTalk (deepcopy_smart_pointer<CAAnimal> mAnimal)
19: {
20:     mAnimal.Talk ();
21: }
```

### ▼ 分析:

可以看到, `deepcopy_smart_pointer` 在第 10~15 行实现了一个复制构造函数, 使得能够通过 `Clone` 函数对多态对象进行深度复制——对象必须实现 `Clone` 函数。为简单起见, 这里假设基类 `CAAnimal` 实现的虚函数为 `Clone`。通常, 实现深度复制模型的智能指针通过模板参数或函数对象提供该函数。

下面是 `deepcopy_smart_pointer` 的一种用法:

```
deepcopy_smart_pointer <CAAnimal> pDog (new CDog());
MakeAnimalTalk (pDog);    // No slicing issues as pDog is deep-copied
```

```
deepcopy_smart_pointer <CAAnimal> pAnimal (new CCat());
MakeAnimalTalk (pCat);    // No slicing
```

因此, 当智能指针作为指向基类 `CAAnimal` 的指针被传递时, 其构造函数实现的深度复制将发挥作用, 确保传递的对象不会出现切片问题——虽然从语法上说, 目标函数 `MakeAnimalTalk()` 只要求基类部分。

基于深度复制的机制的不足之处在于性能。对有些应用程序来说, 这可能不是问题, 但对于其他很多应用程序来说, 这可能导致程序员可不使用智能指针, 而将指向基类的指针 (常规指针 `CAAnimal*`) 传递给函数, 如 `MakeAnimalTalk()`。其他指针类型以各种方式试图解决这种性能问题。

## 26.3.2 写时复制机制

写时复制机制 (Copy on Write, COW) 试图对深度复制智能指针的性能进行优化, 它共享指针, 直到首次写入对象。首次调用非 `const` 函数时, COW 指针通常为该非 `const` 函数操作的对象创建一个副本, 而其他指针实例仍共享源对象。

COW 深受很多程序员的喜欢。实现 `const` 和非 `const` 版本的运算符 `*` 和 `->`, 是实现 COW 指针功能的关键。非 `const` 版本用于创建副本。

重要的是, 选择 COW 指针时, 在使用这样的实现前务必理解其实现细节。否则, 复制时将出现复制得太少或太多的情况。

## 26.3.3 引用计数智能指针

引用计数是一种记录对象的用户数量的机制。当计数降低到零后, 便将对象释放。因此, 引用计数提供了一种优良的机制, 使得可共享对象而无法对其进行复制。如果读者使用过微软的 COM 技术, 肯定知道引用计数的概念。

这种智能指针被复制时, 需要将对象的引用计数加 1。至少有两种常用的方法来跟踪计数:

- 在对象中维护引用计数;
- 引用计数由共享对象中的指针类维护。

前者称为入侵式引用计数, 因为需要修改对象以维护和递增引用计数并将其提供给管理对象的智能指针。COM 采取的就是这种方法。后者是智能指针类将计数保存在自由存储区 (如动态分配的整型), 复制时复制构造函数将这个值加 1。

因此, 使用引用计数机制, 程序员只应通过智能指针来处理对象。在使用智能指针管理对象的同时让原始指针指向它是一种糟糕的做法, 因为智能指针将在它维护的引用计数减为零时释放对象, 而原始指针将继续指向已不属于当前应用程序的内存。引用计数还有一个独特的问题: 如果两个对象分别存储指向对方的指针, 这两个对象将永远不会被释放, 因为它们的生命周期依赖性导致其引用计数

最少为1。

### 26.3.4 引用链接智能指针

引用链接智能指针不主动维护对象的引用计数，而只需知道计数什么时候变为零，以便能够释放对象。

之所以称为引用链接，是因为其实现是基于双向链表的。通过复制智能指针来创建新智能指针时，新指针将被插入到链表中。当智能指针离开作用域进而被销毁时，析构函数将把它从链表中删除。与引用计数的指针一样，引用链接指针也存在生命周期依赖性导致的问题。

### 26.3.5 破坏性复制

破坏性复制是这样一种机制，即在智能指针被复制时，将对象的所有权转交给目标指针并重置原来的指针。

```
destructive_copy_smartptr <SomeClass> pSmartPtr (new SomeClass ());
```

```
SomeFunc (pSmartPtr);    // Ownership transferred to SomeFunc
// Don't use pSmartPtr in the caller any more!
```

虽然破坏性复制机制使用起来并不直观，但它有一个优点，即可确保任何时刻只有一个活动指针指向对象。因此，它非常适合从函数返回指针以及需要利用其“破坏性”的情形。

`std::auto_ptr` 是最流行（也可以说是最臭名昭著，取决于您如何看）的破坏性复制指针。前面的代码演示了使用这种指针的缺点，它还表明，当这种智能指针被传递给函数或复制给另一个指针时，它就没有用了。程序清单 26.3 是一种破坏性复制指针的实现，它没有采用推荐的标准 C++ 编程方法。

程序清单 26.3 一个破坏性复制智能指针

```
1: template <typename T>
2: class destructivecopy_pointer
3: {
4: private:
5:     T* m_pObject;
6: public:
7:     // other members, constructors, destructors, operators* and ->, etc...
8:
9:     // copy constructor
10:    destructivecopy_pointer(destructivecopy_pointer& source)
11:    {
12:        // Take ownership on copy
13:        m_pObject = source.m_pObject;
14:
15:        // destroy source
16:        source.m_pObject = 0;
17:    }
18:
19:    // assignment operator
20:    destructivecopy_pointer& operator= (destructivecopy_pointer& rhs)
21:    {
22:        if (m_pObject != source.m_pObject)
23:        {
24:            delete m_pObject;
25:            m_pObject = source.m_pObject;
26:            source.m_pObject = 0;
27:        }
28:    }
29: };
```

#### ▼ 分析：

程序清单 26.3 演示了基于破坏性复制的智能指针实现的最重要部分。第 10~17 行和第 20~28 行

包含复制构造函数和赋值运算符。从中可知，这些函数实际上使源指针在复制后失效，即复制构造函数在复制后将源指针置为零，这就是“破坏性复制”的由来。赋值运算符执行的操作与此相同。

这几行代码对破坏性复制智能指针的实现来说至关重要，但也深受诟病。可以看到，与大多数 C++ 实现不同，该智能指针类的复制构造函数和赋值运算符不能接受 const 引用，其原因显而易见：它在复制输入引用后使其无效。这不符合传统复制构造函数与赋值运算符的语义，因此 C++ 原旨主义者不喜欢这种智能指针类。

鉴于这种智能指针销毁源引用，这也使得它不适合用于 STL 容器，如 std::vector 或任何其他动态集合类。这些容器需要在内部复制内容，这将导致指针失效。

由于种种原因，很多程序员像躲避瘟疫一样避免使用破坏性复制智能指针。然而，最流行的一种智能指针实现——std::auto\_ptr 就是这种类型的，它是标准模板库的一部分，因此程序员至少必须了解其工作原理。

## 26.4 使用 std::auto\_ptr

auto\_ptr 是一种基于破坏性复制的智能指针，它在复制时交出对象的拥有权，并在离开作用域时释放（销毁）其拥有的对象。

要使用 std::auto\_ptr，必须包含如下头文件：

```
#include <memory>
```

为研究使用 std::auto\_ptr 的效果，下面创建示例类 CSome，它只是在构造函数和析构函数中打印几行信息以说明其生命周期：

```
#include <iostream>
```

```
class CSomeClass
{
public:
    // Constructor
    CSomeClass() {std::cout << "CSomeClass: Constructed!" << std::endl;}

    ~CSomeClass() {std::cout << "CSomeClass: Destructed!" << std::endl;}

    void SaySomething () {std::cout << "CSomeClass: Hello!" << std::endl;}
};
```

下面使用一个针对它的 auto\_ptr 对象，如程序清单 26.4 所示。

程序清单 26.4 使用 std::auto\_ptr

```
1: #include <memory>
2: void UsePointer (std::auto_ptr <CSomeClass> spObj);
3:
4: int main ()
5: {
6:     using namespace std;
7:     cout << "main() started" << endl;
8:
9:     auto_ptr <CSomeClass> spObject (new CSomeClass ());
10:
11:     cout << "main: Calling UsePointer()" << endl;
12:
13:     // Call a function, transfer ownership
14:     UsePointer (spObject);
15:
16:     cout << "main: UsePointer() returned, back in main()" << endl;
17:
18:     // spObject->SaySomething ();    // invalid pointer!
19:
20:     cout << "main() ends" << endl;
21:
22:     return 0;
23: }
```

```

24:
25: void UsePointer (auto_ptr <CSomeClass> spObj)
26: {
27:     cout << "UsePointer: started, will use input pointer now" << endl;
28:
29:     // Use the input pointer
30:     spObj->SaySomething ();
31:
32:     cout << "UsePointer: will return now" << endl;
33: }

```

#### ▼ 输出:

```

main() started
CSomeClass: Constructed!
main: Calling UsePointer()
UsePointer: started, will use input pointer now
CSomeClass: Hello!
UsePointer: will return now
CSomeClass: Destructed!
main: UsePointer() returned, back in main()
main() ends

```

#### ▼ 分析:

从输出可知, 虽然 `spObject` 指向的对象是在 `main()` 中创建的, 但它被自动销毁, 而无需调用 `delete` 运算符。另一个需要注意的地方是, 对象是在函数 `UsePointer` 显示 “UsePointer: will return now” 与 `main()` 显示 “main: UsePointer () returned, back in main()” 之间被销毁的。换句话说, 对象是在 `UsePointer()` 返回时被销毁的, 因为此时变量 `spObj` 将离开作用域, 因此被销毁。这是 `auto_ptr` 的行为: 当指针 (这里是 `UsePointer` 提供的 `spObj`) 离开作用域时, 指针拥有的对象也将被销毁。`main()` 中的 `spObject` 不需要释放, 因为在第 13 行复制它时它已失去了对对象的拥有权。

注意, 将第 18 行注释掉了——在 `auto_ptr` 对象被复制或传递给函数后使用它是危险的, 因此复制操作已导致它无效。千万不要这样做!

总之, 程序清单 26.4 演示了使用智能指针的优点 (条件是了解其行为)。智能指针为程序员管理对象的生命周期。在有些情况下, 可使用更复杂的智能指针帮助编写多线程应用程序, 以便能够同步访问动态分配的数据。

## 26.5 流行的智能指针库

显然, C++ 标准库提供的智能指针并不能满足所有程序员的需求, 这就是还有很多其他智能指针库的原因。

Boost ([www.boost.org](http://www.boost.org)) 提供了一些经过测试且文档完善的智能指针类, 还有很多其他的实用类。

同样, 在 Windows 平台上编写 COM 应用程序的程序员应使用 ATL 框架中的智能指针类 (如 `CComPtr` 和 `CComQIPtr`) 来管理 COM 对象, 而不要使用原始界面指针。

## 26.6 总结

本章介绍了使用正确的智能指针有助于编写使用指针的代码, 让程序员无需关心内存分配和对象拥有权等问题。本章还介绍了各种智能指针类型, 并指出在应用程序中使用智能指针类前务必要了解其行为。最后, 还介绍了最流行 (或者说最臭名昭著的) 智能指针 `std::auto_ptr` 及其缺点。

## 26.7 问与答

问：在需要指针向量时，是否应将 `auto_ptr` 作为向量存储的对象类型？

答：不应该。如果这样做，对向量中的元素执行复制或赋值操作将导致对象不可用。

问：要成为智能指针类，类需要实现哪两个运算符？

答：运算符 `*` 和 `->`，这两个运算符使得可像使用常规指针那样使用类的对象。

问：假设有一个应用程序，其中的 `CClass1` 和 `CClass2` 分别包含一个指向 `CClass2` 对象和 `CClass1` 对象的成员属性。在这种情况下，是否应使用引用计数指针？

答：不应该。由于生命周期依赖性，引用计数将不会减少到零，导致两个类的对象永久性地留在堆中。

问：智能指针有多少种？

答：成千上万，甚至数百万。程序员应只使用文档完善且来源可靠（如来自 Boost）的智能指针。

问：`string` 类也在堆中动态地管理字符数组，它也是智能指针吗？

答：不是。`string` 类通常没有实现运算符 `*` 和 `->`，因此不属于智能指针。

## 26.8 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 26.8.1 测试

1. 为应用程序编写自己的智能指针前应查看什么地方？
2. 智能指针是否会严重降低应用程序的性能？
3. 引用计数智能指针在什么地方存储引用计数数据？
4. 引用链接指针使用的链表机制是单向链表还是双向链表？

### 26.8.2 练习

1. 排错：指出下述代码中的错误：

```
std::auto_ptr<CSomeClass> pObject (new CSomeClass ());  
std::auto_ptr<CSomeClass> pAnotherObject (pObject);  
pObject->DoSomething ();  
pAnotherObject->DoSomething();
```

2. 使用 `auto_ptr` 类实例化一个 `CDog` 对象，`CDog` 类继承了 `CAnimal` 类。将该对象作为 `CAnimal` 指针传递时是否会出现切片问题。



# 第 27 章

## 处理流

到目前为止，我们一直使用 `cout` 将数据写到屏幕，使用 `cin` 从键盘读取数据，却对它们的工作原理没有全面了解。本章将全面介绍 `cout` 和 `cin`。

在本章中，您将学习：

- 什么是流，如何使用它们
- 如何使用流来管理输入和输出
- 如何使用流来读写文件

### 27.1 流概述

C++ 没有定义如何将数据写入屏幕或文件，也没有定义如何将数据读入程序。然而，使用 C++ 编写程序时，这些是不可缺少的部分，标准 C++ 库包含用来方便输入和输出 (I/O) 的 `iostream` 库。

将输入和输出同语言分开并使用库来处理输入和输出的优点是，更容易使语言独立于平台。也就是说，可以在 PC 上编写 C++ 程序，然后在 Sun 工作站上重新编译并运行它们；或者在 Linux 上重新编译使用 Windows C++ 编译器创建的代码，然后运行它。编译器厂商只需提供正确的库便万事大吉。至少从理论上说是这样的。

#### 注意

库是一组扩展名为 `.obj` 或 `.o` 的文件，可将它们链接到程序以提供额外的功能。这是最基本的代码重用形式，从古老的程序员将 0 和 1 凿刻到洞穴壁上一直沿用至今。

当前，除对文件输入外，流对 C++ 编程来说不那么重要了。C++ 程序已经发展到使用操作系统或编译器厂商提供的图形用户界面 (GUI) 来同屏幕、文件和用户交互。这包括 Windows 库、X Windows System 库、微软 Foundation Classes 以及 Borland 的 Windows 和 X Windows System 用户界面的 Kylix 抽象。由于这些库是专门针对操作系统的，并非 C++ 标准的组成部分，因此本书不讨论它们。

流是 C++ 标准的组成部分，因此本章将讨论它们。另外，为理解输入和输出的内部工作原理，最好理解流。然而，读者还应对操作系统或厂商提供的 GUI 库有大致了解。

#### 27.1.1 数据流的封装

文本输入和输出可使用 `iostream` 类来完成。`Iostream` 将数据流视为一个字节接一个字节流的流。如果流的目标是文件或控制台屏幕，数据源通常是程序的一部分。如果流的方向与此相反，数据将来自键盘或磁盘文件并流入到数据变量中。

流的主要目标是，将从磁盘读取文件或将输入写入控制台屏幕的问题封装起来。创建流后，程序就可以使用它，流将负责处理所有的细节。图 27.1 说明了这种基本思想。



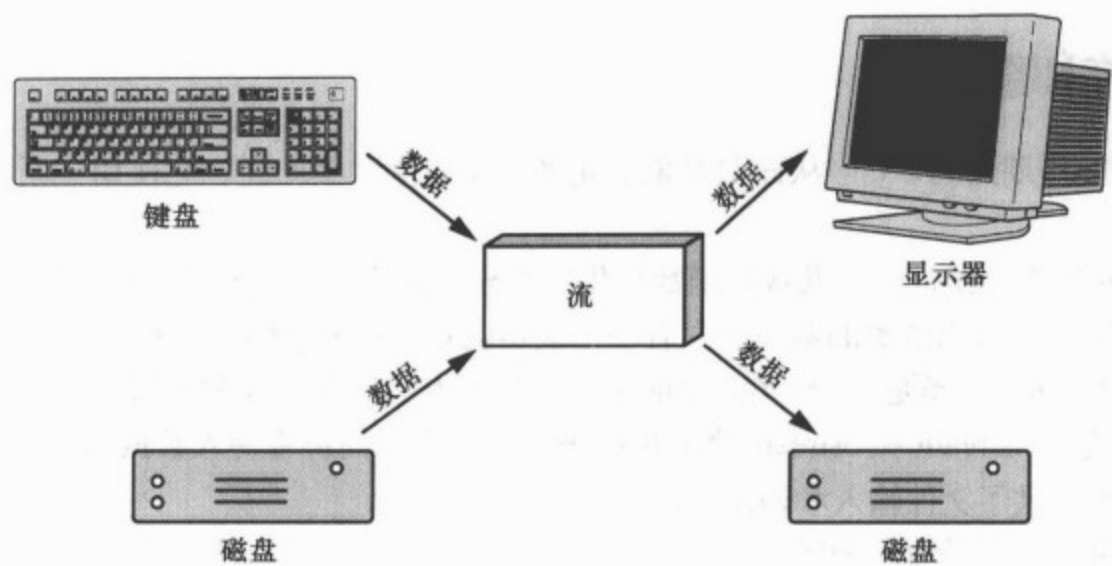


图 27.1 通过流进行封装

27.1.2 理解缓冲技术

将数据写入磁盘（和屏幕）是非常“昂贵”的。相对而言，将数据写入磁盘或从磁盘读取数据都要花很长的时间，读写磁盘可能导致程序暂停执行。为解决这个问题，流提供了缓冲技术。使用缓冲技术时，数据被写入到流中，而不立刻写入到磁盘中。不断地填充流的缓冲区，当缓冲区填满后，一次性将其中的所有数据写入磁盘。

这类似于不断向水箱里注水，但水不从底部流出来。图 27.2 说明了这个概念。  
当水（数据）达到水箱顶部时，阀门打开，所有的水喷涌而出，如图 27.3 所示。

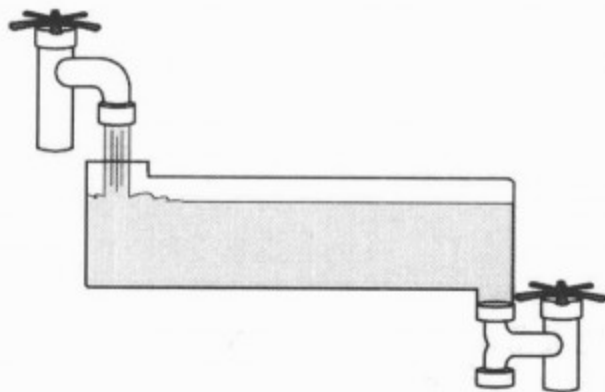


图 27.2 填充缓冲区

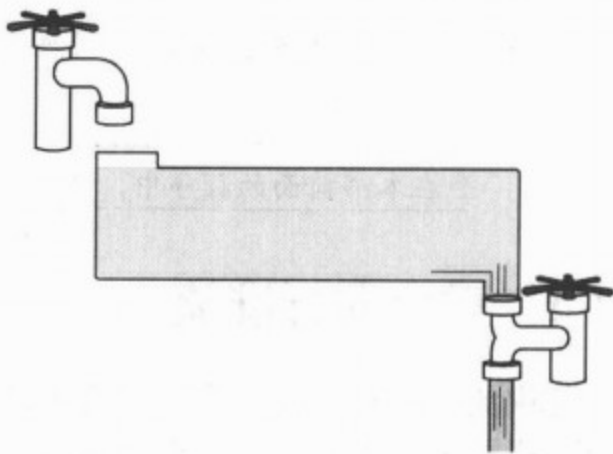


图 27.3 清空缓冲区

清空缓冲区后，底部阀门关上，顶部阀门打开，然后更多的水流入缓冲水箱，如图 27.4 所示。  
偶尔也需要在水箱未注满前就放掉水箱中的水，这称为刷新缓冲区，如图 27.5 所示。

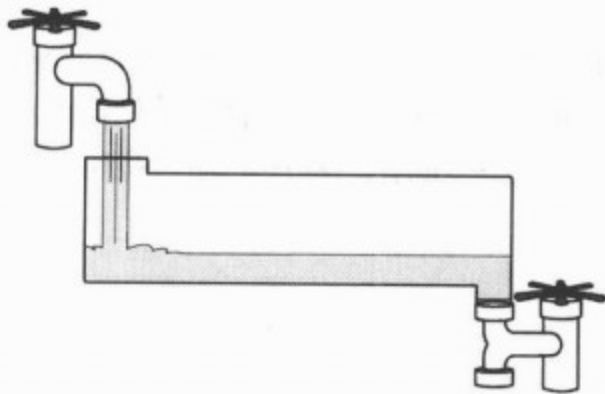


图 27.4 重新填充缓冲区

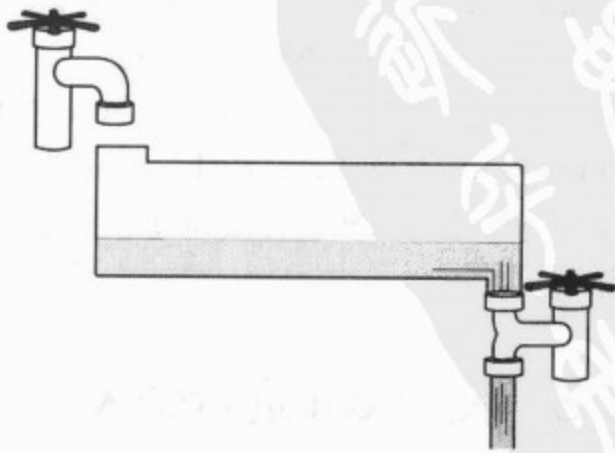


图 27.5 刷新缓冲区

## 27.2 流和缓冲区

正如读者可能预期到的，C++从面向对象的角度来实现流和缓冲区。它使用一系列的类和对象来完成这项任务。

- streambuf 类管理缓冲区，其成员函数提供了填充、清空、刷新和处理缓冲区的其他功能。
- ios 类是输入和输出流类的基类，它有一个成员变量为 streambuf 对象。
- istream 和 ostream 类是从 ios 类派生而来的，用于专门管理输入和输出行为。
- iostream 是从 istream 和 ostream 类派生而来的，提供了向屏幕写入数据的输入和输出方法。
- fstream 类提供了文件输入和输出功能。

本章后面将更详细地介绍这些类。

## 27.3 标准 I/O 对象

包含 iostream 类的 C++ 程序启动时，将创建并初始化 4 个对象。

- cin: 处理来自标准输入设备（键盘）的输入。
- cout: 处理到标准输出设备（控制台屏幕）的输出。
- cerr: 处理到标准错误设备（控制台屏幕）的非缓冲输出。由于这是非缓冲的，因此发送到 cerr 的任何数据都将立即写入标准错误设备，而不要等到缓冲区填满或收到刷新命令。
- clog: 处理输出到标准错误设备（控制台屏幕）的缓冲错误信息。这种输出通常被重定向到日志文件，这将在下一节介绍。

### 注意

编译器自动将 iostream 类库添加到程序中。要使用这些功能，只需在程序开头加入正确的 include 语句：

```
#include <iostream>
```

在本书前面的程序中，您一直在这样做。

## 27.4 重定向标准流

每种标准流（输入、输出和错误）都可以重定向到其他设备。标准错误流（cerr）经常被重定向到文件，而标准输入流（cin）和输出流（cout）可使用操作系统命令重定向到文件。

### 注意

重定向指的是将输出（或输入）发送到不同于默认设备的地方，而管道技术（piping）指的是将一个程序的输出用作另一个程序的输入。

重定向是一种操作系统功能，而不是 iostream 库的功能。C++ 只提供了访问 4 种标准设备的途径，要重定向到其他设备，需要用户去完成。

DOS（Windows 命令提示符）和 Unix 的重定向运算符是 <（重定向输入）和 >（重定向输出）。与 DOS（Windows 命令提示符）相比，Unix 提供了更高级的重定向功能，但基本思想是相同的：将本来发送到屏幕的输出写入到文件或通过管道将其提供给另一个程序。同样，程序的输入也可从文件而不是从键盘读取。

## 27.5 使用 cin 进行输入

全局对象 cin 负责输入，在程序中包含 iostream 后便可使用它。在以前的示例中，您使用重载的提

取运算符 (>>) 将数据存储到程序变量中。这是如何工作的呢？您可能还记得，语法是这样的：

```
int someVariable;
cout << "Enter a number: ";
cin >> someVariable;
```

全局对象 cout 将在本章后面讨论。现在，将重点放在第 3 行：cin >> someVariable;。有关 cin 您能猜到些什么呢？

显然，它必须是全局对象，因为您没有在自己的代码中定义它。根据以前的运算符使用经验，您知道 cin 重载了提取运算符 (>>)：将存放在 cin 缓冲区中的数据写入到局部变量 someVariable 中。

读者可能还不知道的是，cin 重载了接受各种参数的提取运算符，包括 int&、short&、long&、double&、float&、char&、char\* 等。遇到代码 cin >> someVariable; 时，编译器将判断 someVariable 的类型。在上面的示例中，someVariable 是一个 int 变量，因此调用下面的函数：

```
istream & operator>> (int &)
```

注意，由于参数是按引用传递的，因此提取运算符能够对原始变量进行操作。程序清单 27.1 演示了 cin 的用法。

#### 程序清单 27.1 cin 能处理不同的数据类型

```
1: //Listing 27.1 - character strings and cin
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     int myInt;
8:     long myLong;
9:     double myDouble;
10:    float myFloat;
11:    unsigned int myUnsigned;
12:
13:    cout << "Int: ";
14:    cin >> myInt;
15:    cout << "Long: ";
16:    cin >> myLong;
17:    cout << "Double: ";
18:    cin >> myDouble;
19:    cout << "Float: ";
20:    cin >> myFloat;
21:    cout << "Unsigned: ";
22:    cin >> myUnsigned;
23:
24:    cout << "\n\nInt:\t" << myInt << endl;
25:    cout << "Long:\t" << myLong << endl;
26:    cout << "Double:\t" << myDouble << endl;
27:    cout << "Float:\t" << myFloat << endl;
28:    cout << "Unsigned:\t" << myUnsigned << endl;
29:    return 0;
30: }
```

#### ▼ 输出：

```
int: 2
Long: 70000
Double: 987654321
Float: 3.33
Unsigned: 25

Int: 2
Long: 70000
Double: 9.87654e+008
Float: 3.33
Unsigned: 25
```

#### ▼ 分析：

第 7~11 行声明了各种类型的变量，第 13~22 行提示用户输入这些变量的值，第 24~28 行使用

cout 打印结果。

输出结果表明，值被输入到了“正确”的变量类型中，程序按期望的那样运行。

### 27.5.1 输入字符串

cin 还能够处理字符指针 (char\*) 参数，因此，可创建一个字符缓冲区，并使用 cin 来填充它。例如，可以编写这样的代码：

```
char YourName[50]
cout << "Enter your name: ";
cin >> YourName;
```

如果输入 Jesse，数组 YourName 将包含字符 J、e、s、s、e 和 \0。最后一个为空字符。cin 自动在字符串末尾加上一个空字符，因此缓冲区必须有足够的空间容纳整个字符串和空字符。对于 cin 对象来说，空字符表示字符串结束。

### 27.5.2 字符串的问题

使用 cin 成功完成上述操作后，当您试图输入全名时可能感到惊讶：cin 将空格视为分隔符，因此无法读取全名。遇到空格或换行符后，cin 认为输入结束；如果输入的是字符串，它将加上一个空字符。程序清单 27.2 演示了这种问题。

程序清单 27.2 试图向 cin 写入多个单词

---

```
1: //Listing 27.2 - character strings and cin
2: #include <iostream>
3:
4: int main()
5: {
6:     char YourName[50];
7:     std::cout << "Your first name: ";
8:     std::cin >> YourName;
9:     std::cout << "Here it is: " << YourName << std::endl;
10:    std::cout << "Your entire name: ";
11:    std::cin >> YourName;
12:    std::cout << "Here it is: " << YourName << std::endl;
13:    return 0;
14: }
```

---

#### ▼ 输出：

---

```
Your first name: Jesse
Here it is: Jesse
Your entire name: Jesse Liberty
Here it is: Jesse
```

---

#### ▼ 分析：

第 6 行创建了一个名为 YourName 的字符数组，用于存储用户的输入。第 7 行提示用户输入名，从输出可知，名被正确存储。第 10 行提示用户输入全名。cin 读取输入，遇到姓和名之间的空格时，在第一个词后加上一个空字符并结束输入。这显然不是我们的本意。要理解 cin 为何这样做，请看程序清单 27.3，它演示了多个变量的输入。

程序清单 27.3 输入多个变量

---

```
1: //Listing 27.3 - character strings and cin
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     int myInt;
```

---

```

8:    long myLong;
9:    double myDouble;
10:   float myFloat;
11:   unsigned int myUnsigned;
12:   char myWord[50];
13:
14:   cout << "int: ";
15:   cin >> myInt;
16:   cout << "Long: ";
17:   cin >> myLong;
18:   cout << "Double: ";
19:   cin >> myDouble;
20:   cout << "Float: ";
21:   cin >> myFloat;
22:   cout << "Word: ";
23:   cin >> myWord;
24:   cout << "Unsigned: ";
25:   cin >> myUnsigned;
26:
27:   cout << "\n\nInt:\t" << myInt << endl;
28:   cout << "Long:\t" << myLong << endl;
29:   cout << "Double:\t" << myDouble << endl;
30:   cout << "Float:\t" << myFloat << endl;
31:   cout << "Word: \t" << myWord << endl;
32:   cout << "Unsigned:\t" << myUnsigned << endl;
33:
34:   cout << "\n\nInt, Long, Double, Float, Word, Unsigned: ";
35:   cin >> myInt >> myLong >> myDouble;
36:   cin >> myFloat >> myWord >> myUnsigned;
37:   cout << "\n\nInt:\t" << myInt << endl;
38:   cout << "Long:\t" << myLong << endl;
39:   cout << "Double:\t" << myDouble << endl;
40:   cout << "Float:\t" << myFloat << endl;
41:   cout << "Word: \t" << myWord << endl;
42:   cout << "Unsigned:\t" << myUnsigned << endl;
43:
44:   return 0;
45: }

```

### ▼ 输出:

```

Int: 2
Long: 30303
Double: 393939397834
Float: 3.33
Word: Hello
Unsigned: 85

```

```

Int: 2
Long: 30303
Double: 3.93939e+011
Float: 3.33
Word: Hello
Unsigned: 85

```

```

Int, Long, Double, Float, Word, Unsigned: 3 304938 393847473 6.66 bye -2

```

```

Int: 3
Long: 304938
Double: 3.93847e+008
Float: 6.66
Word: bye
Unsigned: 4294967294

```

### ▼ 分析:

这里也创建了多个变量，这次包括一个 char 数组。提示用户输入，并正确地打印输出。

第 34 行提示用户一次性提供全部输入，然后输入的每个“单词”被赋给相应的变量。为实现这种多重赋值，cin 必须将输入的每个单词视为每个变量的完整输入。如果 cin 认为整个输入是变量输入的一部分，将不能实现这种级联输入。

注意，第42行要求的最后一个对象是无符号整数，但用户输入的是-2。由于 cin 认为它显示的是一个无符号整数，因此按无符号整数计算-2的位模式，这样使用 cout 输出时，显示的是 4 294 967 294。无符号值 4 294 967 294 与有符号值-2的位模式完全相同。

本章后面将讨论如何将包含多个单词的字符串输入到缓冲区。现在的问题是：提取运算符如何处理这种级联方式？

### 27.5.3 >>的返回值

>>的返回值是一个 istream 对象引用。由于 cin 本身是 istream 对象，因此提取操作的返回值可用作下一次提取操作的输入。

```
int varOne, varTwo, varThree;
cout << "Enter three numbers: ";
cin >> varOne >> varTwo >> varThree;
```

对于代码 `cin >> VarOne >> varTwo >> varThree;`，首先执行的提取操作为 `cin >> VarOne`，其返回值是另一个 istream 对象，该对象的提取运算符将变量 varTwo 作为参数，就像代码为：

```
((cin >> varOne) >> varTwo) >> varThree;
```

后面讨论 cout 时将再次使用这种技术。

## 27.6 cin 的其他成员函数

除了重载运算符>>外，cin 还有很多其他成员函数。它们用于对输入进行细致的控制，让您能够：

- 读取单个字符；
- 读取字符串；
- 忽略输入；
- 查看缓冲区中的下一个字符；
- 将数据放回缓冲区。

### 27.6.1 单字符输入

运算符>>接受一个字符引用作为参数，可用于从标准输入中读取单个字符。成员函数 `get()` 也可用于获取单个字符，且有两种调用方式：不提供任何参数并使用其返回值；使用一个字符引用参数来调用。

#### 1. 使用不接受任何参数的 `get()`

`get()` 的第一种形式不接受任何参数，它返回找到的字符值，如果达到文件末尾则返回 EOF（文件末尾）。不接受任何参数的 `get()` 不常用。

这种形式的 `get()` 不能像 cin 那样用于级联多次输入，因为其返回值不是 istream 对象。所以下面的语句是非法的：

```
cin.get() >> myVarOne >> myVarTwo; // illegal
```

`cin.get() >> myVarOne` 的返回值是一个整数，而不是 istream 对象。

程序清单 27.4 演示了不接受参数的 `get()` 的常见用法。

#### 程序清单 27.4 使用不接受参数的 `get()`

```
1:
2: #include <iostream>
3:
4: int main()
5: {
6:     char ch;
```



```

7:   while ( (ch = std::cin.get()) != EOF)
8:   {
9:       std::cout << "ch: " << ch << std::endl;
10:  }
11:  std::cout << "\nDone!\n";
12:  return 0;
13: }

```

**注意**

要退出该程序，必须通过键盘输入文件末尾标记。为此，在 DOS 计算机上按 Ctrl+Z，在 Unix 工作站上按 Ctrl+D。

**▼ 输出:**

```

Hello
ch: H
ch: e
ch: l
ch: l
ch: o
ch:

World
ch: W
ch: o
ch: r
ch: l
ch: d
ch:

^Z (ctrl-z)

Done!

```

**▼ 分析:**

第 6 行声明了一个局部字符变量 `ch`。`while` 循环将 `cin.get()` 读取的输入赋给 `ch`，如果该字符不是 EOF，就打印出它。然而，输出被缓冲，直到遇到换行符。遇到 EOF（在 DOS 机器上按 Ctrl+Z 或在 Unix 机器上按 Ctrl+D）后退出循环。注意，并非每种 `istream` 实现都支持这种版本的 `get()`，虽然它现在是 ANSI/ISO 标准的组成部分。

**2. 使用接受字符引用参数的 `get()`**

用字符变量作为参数来调用 `get()` 时，将把输入流中的下一个字符赋给该字符变量。返回值是一个 `istream` 对象，因此这种形式的 `get()` 可以级联，如程序清单 27.5 所示。

程序清单 27.5 使用接受参数的 `get()`

```

1:
2: #include <iostream>
3:
4: int main()
5: {
6:     char a, b, c;
7:
8:     std::cout << "Enter three letters: ";
9:
10:    std::cin.get(a).get(b).get(c);
11:
12:    std::cout << "a: " << a << "\nb: ";
13:    std::cout << b << "\nc: " << c << std::endl;
14:    return 0;
15: }

```

**▼ 输出:**

```

Enter three letters: one
a: o
b: n
c: e

```

## ▼ 分析:

第6行创建了3个字符变量:a、b和c。第10行以级联方式调用了`cin.get()`3次。首先调用`cin.get(a)`,将第1个字符赋给a并返回`cin`。这样接下来将调用`cin.get(b)`,将下一个字符赋给b。最后,调用`cin.get(c)`,将第3个字符赋给c。

由于`cin.get(a)`的结果为`cin`,因此可以这样编写代码:

```
cin.get(a) >> b;
```

在这种情况下,`cin.get(a)`的结果为`cin`,因此第2部分为`cin>>b`。

应该	不应该
<p>需要跳过空白时,应使用提取运算符(&gt;&gt;)。</p> <p>需要检查包括空白在内的每个字符时,应使用接受一个字符参数的<code>get()</code>。</p>	<p>如果不清楚要做什么,不要级联<code>cin</code>语句来读取多个输入。使用多条<code>cin</code>语句更合适,它比级联<code>cin</code>语句更容易理解。</p>

## 27.6.2 从标准输入读取字符串

要将输入存储到字符数组中,可使用提取运算符(>>)、成员函数`get()`的第3个版本或`getline()`。

`get()`的第3个版本接受3个参数:

```
get( pCharArray, StreamSize, TermChar );
```

第1个参数(`pCharArray`)是字符数组指针,第2个参数(`StreamSize`)是要读取的最大字符数加1,第3个参数(`TermChar`)是结束字符。如果第2个参数为20,`get()`将读取19个字符,然后在存储在第1个参数中的字符串末尾加上空字符。第3个参数(结束字符)默认为换行符('\n')。如果`get()`在读取最大字符数之前遇到结束字符,将添加空字符,并将结束字符留在缓冲区中。

程序清单 27.6 演示了这种格式的`get()`的用法。

程序清单 27.6 使用接受字符数组作为参数的`get()`

```

1:
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     char stringOne[256];
8:     char stringTwo[256];
9:
10:    cout << "Enter string one: ";
11:    cin.get(stringOne,256);
12:    cout << "stringOne: " << stringOne << endl;
13:
14:    cout << "Enter string two: ";
15:    cin >> stringTwo;
16:    cout << "StringTwo: " << stringTwo << endl;
17:    return 0;
18: }
```

## ▼ 输出:

```

Enter string one: Now is the time
stringOne: Now is the time
Enter string two: For all good
StringTwo: For
```

## ▼ 分析:

第7行和第8行创建了2个字符数组。第10行提示用户输入一个字符串,第11行调用了`cin.get()`。第1个参数是要填充的缓冲区,第2个参数是`get()`将读取的最大字符数加1(多出来的那个位置用于

存储空字符 ('\0')), 第 3 个参数默认为换行符。

用户输入 Now is the time。由于用户以换行符结束该短语, 因此该短语被存储到 stringOne 中, 并在末尾加上一个空字符。

第 14 行提示用户再输入一个字符串, 这次使用提取运算符 (>>) 来读取。由于提取运算符读取第 1 个空白字符前的所有字符, 因此第 2 个字符串中存储的是 “For” 和空字符, 这显然不是我们的本意。

接受 3 个参数的 get() 非常适合用来读取字符串, 然而这并非唯一的解决方案。解决这种问题的另一种方法是使用 getline(), 如程序清单 27.7 所示。

程序清单 27.7 使用 getline()

```
1:
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     char stringOne[256];
8:     char stringTwo[256];
9:     char stringThree[256];
10:
11:     cout << "Enter string one: ";
12:     cin.getline(stringOne, 256);
13:     cout << "stringOne: " << stringOne << endl;
14:
15:     cout << "Enter string two: ";
16:     cin >> stringTwo;
17:     cout << "stringTwo: " << stringTwo << endl;
18:
19:     cout << "Enter string three: ";
20:     cin.getline(stringThree, 256);
21:     cout << "stringThree: " << stringThree << endl;
22:     return 0;
23: }
```

#### ▼ 输出:

```
Enter string one: one two three
stringOne: one two three
Enter string two: four five six
stringTwo: four
Enter string three: stringThree: five six
```

#### ▼ 分析:

这个示例值得仔细研究, 其中可能有一些令人意外的地方。第 7~9 行声明了 3 个字符数组。

第 11 行提示用户输入一个字符串, 然后使用 getline() 读取它。和 get() 一样, getline() 接受一个缓冲区和最大字符数作为参数; 但与 get() 不同的是, 它读取换行符并将其丢弃, get() 不丢弃换行符, 将其留在输入缓冲区中。

第 15 行再次提示用户输入一个字符串, 这次使用提取运算符来读取它。从输出可知, 用户输入了 four five six, 但只有第一个单词 (four) 被存储到 stringTwo 中。然后程序第 3 次提示用户输入一个字符串 (显示 Enter string three), 并再次调用 getline() 来读取它。由于 five six 还留在输入缓冲区中, 因此 getline() 读取它, 并在遇到换行符后结束, 第 21 行打印存储在 stringThree 中的字符串。

用户根本没有机会输入第 3 个字符串, 因为原来留在输入缓冲区的数据已经满足了第 3 次读取要求。第 16 行对 cin 的调用没有读取输入缓冲区中的全部内容。第 16 行的提取运算符 (>>) 读取到第一个空格为止, 并将第一个单词存储到字符数组中。

#### get() 和 getline()

成员函数 get() 被重载。在第 1 个版本中, 它不接受任何参数并返回读取的字符的值。在第 2 个版本中, 它接受一个字符引用参数, 并按引用返回 istream 对象。

在第3个也最后一个版本中, `get()` 接受一个字符数组、要读取的最大字符数和结束字符(默认为换行符)作为参数。这种版本的 `get()` 将字符读入数组, 直到读入的字符数比指定的最大字符数少1或遇到结束字符。遇到结束字符后, `get()` 将该字符留在输入缓冲区中并结束读取。

成员函数 `getline()` 也接受3个参数: 要填充的缓冲区、要读取的最大字符数加1以及结束字符。`getline()` 函数使用这3个参数的方式和 `get()` 相同, 只是它丢弃结束字符。

### 27.6.3 使用 `cin.ignore()`

有时可能需要忽略行尾(EOL)或文件尾(EOF)之前的剩余字符, 成员函数 `ignore()` 提供了这种功能。它接受2个参数: 要忽略的最大字符数和结束字符。代码 `ignore(80, '\n')` 将最多忽略80个字符, 直到遇到换行符。然后丢弃换行符, `ignore()` 语句结束。程序清单27.8演示了 `ignore()` 的用法。

程序清单 27.8 使用 `ignore()`

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     char stringOne[255];
7:     char stringTwo[255];
8:
9:     cout << "Enter string one:";
10:    cin.get(stringOne, 255);
11:    cout << "String one: " << stringOne << endl;
12:
13:    cout << "Enter string two: ";
14:    cin.getline(stringTwo, 255);
15:    cout << "String two: " << stringTwo << endl;
16:
17:    cout << "\n\nNow try again...\n";
18:
19:    cout << "Enter string one: ";
20:    cin.get(stringOne, 255);
21:    cout << "String one: " << stringOne << endl;
22:
23:    cin.ignore(255, '\n');
24:
25:    cout << "Enter string two: ";
26:    cin.getline(stringTwo, 255);
27:    cout << "String Two: " << stringTwo << endl;
28:    return 0;
29: }
```

#### ▼ 输出:

```
Enter string one:once upon a time
String one: once upon a time
Enter string two: String two:

Now try again...
Enter string one: once upon a time
String one: once upon a time
Enter string two: there was a
String Two: there was a
```

#### ▼ 分析:

第6行和第7行创建了2个字符数组。第9行提示用户输入, 用户输入 `once upon a time` 后按回车键。第10行使用 `get()` 来读取该字符串。`get()` 将换行符之前的内容存储到 `stringOne` 中, 并将换行符留在输入缓冲区中。

第13行再次提示用户输入, 但第14行使用 `getline()` 来读取缓冲区中换行符之前的内容。由于之前调用 `get()` 时将一个换行符留在了缓冲区中, 因此用户还未输入任何内容之前, 第14行已经结束。

第19行再次提示用户输入, 用户的输入与第一次完全相同。然而这次使用 `ignore()` 来删除换行符



以清空输入流（第 23 行）。这样，当程序执行到第 26 行的 `getline()` 调用时，输入缓冲区是空的，因此用户能够输入下一行字符串。

### 27.6.4 查看和插入字符：`peek()`和 `putback()`

输入对象 `cin` 有 2 个使用起来非常方便的方法：`peek()`和 `putback()`。`peek()`查看但不提取下一个字符，`putback()`将一个字符插入到输入流中。程序清单 27.9 演示了它们的用法。

程序清单 27.9 使用 `peek()`和 `putback()`

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     char ch;
7:     cout << "enter a phrase: ";
8:     while ( cin.get(ch) != 0 )
9:     {
10:         if (ch == '!')
11:             cin.putback('$');
12:         else
13:             cout << ch;
14:             while (cin.peek() == '#')
15:                 cin.ignore(1, '#');
16:     }
17:     return 0;
18: }
```

#### ▼ 输出：

```
enter a phrase: Now!is#the!time#for!fun#l
Now$isthe$timefor$fun$
```

#### ▼ 分析：

第 6 行声明了一个字符变量 `ch`，第 7 行提示用户输入一条短语。该程序旨在将感叹号 (!) 替换为美元符号 (\$) 并删除所有的英镑符号 (#)。

遇到文件末尾字符（在 Windows 计算机上为 `Ctrl + C`，在其他操作系统上为 `Ctrl + Z` 或 `Ctrl + D`）之前，将不断执行第 8~16 行的循环（`cin.get()`到达文件末尾后，将返回 0）。

如果当前字符为感叹号，则将其丢弃，并将美元符号 (\$) 插入输入缓冲区，下次循环将读取该符号。如果当前字符不是感叹号，则将其打印出来（第 13 行）。

第 14 行查看下一个字符，如果是英镑符号，则使用 `ignore()` 方法将其删除，如第 15 行所示。

虽然该程序并非处理这些问题的最有效方式（同时，如果第一个字符为 #，该程序将发现不了），但它演示了这些方法的工作原理。

#### 提示

`peek()`和 `putback()`通常用于分析字符串和其他数据，如编写编译器时。

## 27.7 使用 cout 进行输出

您使用过 `cout` 和重载的插入运算符 (<<) 将字符串、整数和其他数值写入到屏幕。还可以对数据进行格式化：对齐列以及以十进制和十六进制书写数值数据，本节介绍如何完成这种任务。

### 27.7.1 刷新输出

正如您知道的，`endl` 写入一个换行符并刷新输出缓冲区。`endl` 调用 `cout` 的成员函数 `flush()`，后者

输出被缓冲的所有数据。也可以直接调用 `flush()`：

- `cout << flush();`
- 需要清空输出缓冲区并将其中的内容写入屏幕时，该方法非常方便。

## 27.7.2 执行输出的函数

就像 `get()` 和 `getline()` 是提取运算符的补充一样，`put()` 和 `write()` 是插入运算符的补充。

### 1. 使用 `put()` 写入字符

函数 `put()` 用于将单个字符写入输出设备。`put()` 返回一个 `ostream` 引用，而 `cout` 是一个 `ostream` 对象，因此可以像级联插入运算符一样级联 `put()`，如程序清单 27.10 所示。

程序清单 27.10 使用 `put()`

```
1:
2: #include <iostream>
3:
4: int main()
5: {
6:     std::cout.put('H').put('e').put('l').put('l').put('o').put('\n');
7:     return 0;
8: }
```

#### ▼ 输出：

Hello

#### 注意

有些非标准编译器可能不支持上述打印代码。如果您的编译器不能打印单词 Hello，建议跳过该程序清单。

#### ▼ 分析：

第 6 行的执行过程是这样的：`cout.put('H')` 将字母 H 写入屏幕并返回一个 `cout` 对象。这样，还未执行的代码如下：

```
cout.put('e').put('l').put('l').put('o').put('\n');
```

然后写入字母 e 并返回一个 `cout` 对象。这样，还未执行的代码如下：

```
cout.put('l').put('l').put('o').put('\n');
```

重复上述过程：将每个字母写入屏幕并返回 `cout` 对象。将最后一个字符（`'\n'`）写入屏幕后函数返回。

### 2. 使用 `write()` 写入更多字符

函数 `write()` 的工作原理和插入运算符（`<<`）相同，只是它接受一个指出最多写入多少个字符的参数：

```
cout.write(Text, Size)
```

`write()` 的第一个参数是要打印的文本，第二个参数（`Size`）指定要打印 `Text` 中的多少个字符。注意，`Size` 可能大于或小于 `Text` 的实际长度。如果大于，将输出内存中 `Text` 后面的值。程序清单 27.11 演示了 `write()` 的用法。

程序清单 27.11 使用 `write()`

```
1: #include <iostream>
2: #include <string.h>
3: using namespace std;
4:
5: int main()
6: {
7:     char One[] = "One if by land";
8:
9:     int fullLength = strlen(One);
10:    int tooShort = fullLength - 4;
```



```

11:     int tooLong = fullLength + 6;
12:
13:     cout.write(One,fullLength) << endl;
14:     cout.write(One,tooShort) << endl;
15:     cout.write(One,tooLong) << endl;
16:     return 0;
17: }

```

#### ▼ 输出:

```

One if by land
One if by
One if by land i?!

```

#### 注意

在您的计算机上,最后一行的输出可能与此不同,因为访问了未被初始化的变量占用的内存。

#### ▼ 分析:

该程序清单打印一个短语的内容,它每次打印的内容量不同。第 7 行创建了一个短语。第 9 行使用全局方法 `strlen()` 将 `int` 变量 `fullLength` 设置为短语的长度,该方法是通过第 2 行的编译指令包含进来的。`tooShort` 被设置为短语的长度减 4, `tooLong` 被设置为短语的长度加 6。

第 13 行使用 `write()` 打印整个短语。长度被设置为短语的实际长度,因此打印了正确的短语。

第 14 行再次打印,但输出表明,比整个短语少 4 个字符。

第 15 行再次打印,但这次让 `write()` 多写入 6 个字符。写入该短语后,继续写入后续内存中的 6 个字节。该内存中的内容是不确定的,因此读者的输出可能与前面列出的不同。

### 27.7.3 控制符、标记和格式化指令

输出流包含很多状态标记,用于指定计数方式(十进制或十六进制)、字段宽度和字段填充字符。状态标记长 1 字节,其中的每一位都有特殊含义。用这种方式操纵位在第 25 章和第 29 章讨论。每个 `ostream` 标记都可以用成员函数和控制符来设置。

#### 1. 使用 `cout.width()`

输出的默认宽度为刚好能够容纳输出缓冲区中的数字、字符或字符串。可以使用 `width()` 来修改默认宽度设置。

`width()` 是成员函数,必须通过 `cout` 对象来调用。它只修改下一个输出字段的宽度,然后字段宽度设置立刻恢复到默认值。程序清单 27.12 演示了 `width()` 的用法。

程序清单 27.12 调整输出的宽度

```

1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Start >";
7:     cout.width(25);
8:     cout << 123 << "< End\n";
9:
10:    cout << "Start >";
11:    cout.width(25);
12:    cout << 123 << "< Next >";
13:    cout << 456 << "< End\n";
14:
15:    cout << "Start >";
16:    cout.width(4);
17:    cout << 123456 << "< End\n";

```

```
18:
19:     return 0;
20: }
```

▼ 输出:

```
Start >                123< End
Start >                123< Next >456< End
Start >123456< End
```

▼ 分析:

第 1 次输出 (第 6~8 行) 在宽度被设置为 25 (第 7 行) 的字段中打印数字 123, 如第 1 行输出所示。  
第 2 次输出首先在宽度被设置为 25 的字段中打印 123, 然后打印 456。注意, 456 被打印在一个宽度刚刚好的字段中。正如前面指出的, width() 的作用仅持续到下一次输出。  
最后一次输出表明, 将宽度设置为小于输出内容与设置为刚刚好等效。当宽度被设置为比输出内容小时, 不会导致输出被截短。

2. 设置填充字符

通常情况下, cout 用空格填充字段, 如前面所示。有时可能想用其他字符 (如星号) 来填充。为此, 可调用 fill(), 并将要用来填充的字符作为参数传递给它, 如程序清单 27.13 所示。

程序清单 27.13 使用 fill()

```
1:
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     cout << "Start >";
8:     cout.width(25);
9:     cout << 123 << "< End\n";
10:
11:     cout << "Start >";
12:     cout.width(25);
13:     cout.fill('*');
14:     cout << 123 << "< End\n";
15:
16:     cout << "Start >";
17:     cout.width(25);
18:     cout << 456 << "< End\n";
19:
20:     return 0;
21: }
```

▼ 输出:

```
Start >                123< End
Start >*****123< End
Start >*****456< End
```

▼ 分析:

第 7~9 行的功能与前一个示例中完全相同: 在一个宽度为 25 的字段中打印 123。第 11~14 行重复这种操作, 但第 13 行将填充字符设置为星号, 如输出所示。需要注意的是, 不同于函数 width() 那样只对下一次输出有效, fill() 设置的填充字符将一直有效, 直到您修改为止。第 16~18 行的输出证明了这一点。

3. 管理输出状态: 设置标记

当对象的部分或全部数据成员表示可在程序执行期间修改的条件时, 该对象被认为是有状态的。例如, 可以设置是否显示末尾的零 (以免 20.00 被截短为 20)。

Iostream 对象使用标记来记录其状态。可以调用 `setf()` 并传递一个预定义的枚举常量来设置这些标记。例如，要显示末尾的零，可调用 `setf(ios::showpoint)`。

这些枚举常量的作用域为 `iostream` 类 (`ios`)，因此将其作为 `setf()` 的参数时，需要采用全限定方式 `ios::flagname`，如 `ios::showpoint`。表 27.1 列出了一些可使用的标记。要使用这些标记，必须在程序中包含 `iostream`。另外，要使用那些需要参数的标记，还必须包含 `iomanip`。

表 27.1 一些 iostream 设置标记

标记	用途	标记	用途
showpoint	根据精度设置显示小数点和末尾的零	fixed	以小数点表示法显示浮点数
showpos	在正数前面加上正号 (+)	showbase	在十六进制数前加上 0x，指出这是十六进制值
left	让输出左对齐	Uppercase	在十六进制和科学表示法中使用大写字母
right	让输出右对齐	dec	以十进制方式显示
internal	让符号左对齐并让数值右对齐	oct	以八进制方式显示
scientific	用科学表示法显示浮点数	hex	以十六进制方式显示

表 27.1 中的标记也可与插入运算符一起使用。程序清单 27.14 演示了这些设置，还使用了设置宽度的 `setw` 控制符，该控制符也可与插入运算符一起使用。

程序清单 27.14 使用 setf

```
1: #include <iostream>
2: #include <iomanip>
3: using namespace std;
4:
5: int main()
6: {
7:     const int number = 185;
8:     cout << "The number is " << number << endl;
9:
10:    cout << "The number is " << hex << number << endl;
11:
12:    cout.setf(ios::showbase);
13:    cout << "The number is " << hex << number << endl;
14:
15:    cout << "The number is " ;
16:    cout.width(10);
17:    cout << hex << number << endl;
18:
19:    cout << "The number is " ;
20:    cout.width(10);
21:    cout.setf(ios::left);
22:    cout << hex << number << endl;
23:
24:    cout << "The number is " ;
25:    cout.width(10);
26:    cout.setf(ios::internal);
27:    cout << hex << number << endl;
28:
29:    cout << "The number is " << setw(10) << hex << number << endl;
30:    return 0;
31: }
```

▼ 输出:

```
The number is 185
The number is b9
The number is 0xb9
The number is      0xb9
The number is 0xb9
The number is 0x      b9
The number is:0x      b9
```

▼ 分析:

第 7 行将 int 常量 number 初始化为 185，第 8 行按正常方式显示这个值。  
第 10 行再次显示这个值，但使用了控制符 hex，将这个值以十六进制方式显示为 b9。

注意

在十六进制中，b 表示 11； $11 \times 16 = 176$ ， $176 + 9 = 185$ 。

第 12 行设置标记 showbase，这导致在所有十六进制数前加上 0x，如输出所示。  
第 16 行将宽度设置为 10，并采用默认的右对齐。  
第 20 行再次将宽度设置为 10，但设置为左对齐，因此数字被打印在最左边。  
第 25 行再次将宽度设置为 10，但采用两端对齐。因此 0x 被打印在最左边，而 b9 被打印在最右边。  
最后，第 29 行结合使用插入运算符和 setw()，将宽度设置为 10，并再次打印这个值。  
从这个程序清单可知，与插入运算符一起使用时，无需对标记进行全限定：要采用十六进制表示只需使用 hex 即可。而用作函数 setf() 的参数时，必须对标记进行全限定：要采用十六进制表示必须传递 ios::hex。从第 17 行和第 21 行可以看到这种区别。

27.8 流和 printf() 函数之比较

大多数 C++ 程序还提供了标准 C 语言 I/O 库，其中包括函数 printf()。虽然 printf() 在某些方面比 cout 使用起来更容易，但应尽量避免使用它。

printf() 没有提供类型安全性，很容易无意间命令它将整数当作字符显示或反过来。printf() 也不支持类，因此无法告诉它如何打印类数据，而必须将类成员逐个传递给 printf()。

由于有很多使用 printf() 的代码，因此本节简要地介绍一下 printf() 的用法。然而，建议您在 C++ 程序中使用该函数。

要使用 printf()，务必包含头文件 stdio.h。在最简单的情况下，printf() 将一个格式化字符串作为其第一个参数，并将一系列的值作为其他参数。

格式化字符串是用引号括起的文本和转换说明符。所有转换说明符都必须以百分号 (%) 打头。表 27.2 列出了常见的转换说明符。

表 27.2 常见的转换说明符

说明符	用于	说明符	用于
%s	字符串	%ld	双精度
%d	整型	%f	浮点数
%l	长整型		

每个转换说明符还可以以浮点数的方式指定宽度和精度，其中小数点左边的数字表示总宽度，小数点右边的数字表示浮点数的精度。因此 5%d 表示宽度为 5 位的整数，%15.5 表示宽度为 15 位，其中最后 5 位为小数部分的浮点数。程序清单 27.15 演示了 printf() 的各种用法。

程序清单 27.15 使用 printf() 进行打印

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:     printf("%s", "hello world\n");
6:
7:     char *phrase = "Hello again!\n";
8:     printf("%s", phrase);
9:
10:    int x = 5;
```

```

11:    printf("%d\n",x);
12:
13:    char *phraseTwo = "Here's some values: ";
14:    char *phraseThree = " and also these: ";
15:    int y = 7, z = 35;
16:    long longVar = 98456;
17:    float floatVar = 8.8f;
18:
19:    printf("%s %d %d", phraseTwo, y, z);
20:    printf("%s %ld %f\n",phraseThree,longVar,floatVar);
21:
22:    char *phraseFour = "Formatted: ";
23:    printf("%s %5d %10d %10.5f\n",phraseFour,y,z,floatVar);
24:
25:    return 0;
26: }

```

### ▼ 输出:

```

hello world
Hello again!
5
Here's some values: 7 35 and also these: 98456 8.800000
Formatted:      7      35      8.800000

```

### ▼ 分析:

第 1 条 printf() 语句 (第 5 行) 使用标准格式: 用引号括起的转换说明符 (这里为 %s) 和要插入到转换说明符中的值。%s 表示这是一个字符串, 这里是用引号括起的 hello world。

第 2 条 printf() 语句 (第 8 行) 与第一条相同, 但这次使用的是 char 指针, 而不是用引号括起的字符串。

第 3 条 printf() 语句 (第 11 行) 使用整数转换说明符 (%d), 值为 int 变量 x。

第 4 条 printf() 语句 (第 19 行) 要复杂得多: 同时打印 3 个值。首先是每个转换说明符, 然后用逗号分开的值。第 20 行与第 19 行类似, 但使用的说明符和值不同。

最后, 第 23 行使用了格式说明来指定宽度和精度。正如读者所看到的, 这些使用起来都比控制符更容易。

然而, 正如前面指出的, 这里的缺点在于没有类型检查, 同时 printf() 也不能被声明为类的友元或成员函数。因此要打印类的各个成员数据, 必须在参数列表中将每个存取器方法传递给 printf() 函数。

### FAQ

能总结一下如何控制输出吗?

答: 在 C++ 中, 要格式化输出, 需要结合使用特殊字符、输出控制符和标记。

可使用插入运算符将包含下述特殊字符的输出字符串传递给 cout。

\n: 换行符。

\r: 回车。

\t: 制表符。

\\: 反斜杠。

\ddd (八进制数): ASCII 字符。

\a: 振铃。

例如, 下面的代码振铃、打印错误消息并移到下一个制表位:

```
cout << "\aAn error occurred\t"
```

控制符和 cout 运算符一起使用。要使用接受参数的控制符, 必须在程序中包含 iomanip。

下面是不需要包含头文件 iomanip 就可以使用的控制符。

flush: 刷新输出缓冲区。

endl: 换行并刷新输出缓冲区。



oct: 采用八进制。

dec: 采用十进制。

hex: 采用十六进制。

下面是需要包含头文件 `iomanip` 才能使用的控制符:

`setbase(base)`: 设置计数方式 (0 为十进制, 8 为八进制, 10 为十进制, 16 为十六进制)。

`setw (width)`: 设置最小输出字段宽度。

`setfill(ch)`: 指定了字段宽度时使用的填充字符。

`setprecision (p)`: 设置浮点数的精度。

`setiosflags (f)`: 设置一个或多个 ios 标记。

`resetiosflags (f)`: 重置一个或多个 ios 标记。

例如, 下面的代码将字段宽度设置为 12, 将填充字符设置为 “#”, 将计数方式设置为十六进制, 然后打印变量 `x` 的值, 将一个换行符放入缓冲区并刷新缓冲区:

```
cout << setw(12) << setfill('#') << hex << x << endl;
```

除 `flush`、`endl` 和 `setw` 外, 所有控制符在被修改或程序结束前都一直有效。`setw` 在当前 `cout` 结束后恢复到默认值。

很多标记可用作 `setiosflags` 和 `resetiosflags` 控制符的参数, 前面的表 27.1 列出了它们。

更详细的信息请参阅头文件 `ios` 和编译器文档。

## 27.9 文件输入和输出

在处理来自键盘或硬盘的数据以及输出到屏幕或硬盘的数据方面, 流提供了统一的方法。不管在哪种情况下, 都可以使用插入和提取运算符以及其他相关函数和控制符。要打开和关闭文件, 需要创建 `ifstream` 和 `ofstream` 对象, 这将在接下来的几节中介绍。

### 27.9.1 使用 ofstream

用于读写文件的对象被称为 `ofstream` 对象。这些对象是从前面一直在使用的 `iostream` 对象派生出来。

要开始写入文件, 首先必须创建一个 `ofstream` 对象, 然后将其与磁盘上的文件关联起来。要使用 `ofstream` 对象, 必须在程序中包含头文件 `fstream`。

#### 注意

由于 `fstream` 包含了 `iostream.h`, 因此不需要再显式地包含 `iostream`。

### 27.9.2 条件状态

`iostream` 对象包含报告输入和输出状态的标记。可以使用布尔函数 `eof()`、`bad()`、`fail()` 和 `good()` 来检查这些标记。`iostream` 对象遇到文件末尾 (EOF) 时, 函数 `eof()` 返回 `true`; 如果您试图进行非法操作, 函数 `bad()` 将返回 `true`; 在 `bad()` 返回 `true` 或操作失败时, 函数 `fail()` 将返回 `true`; 最后, 在其他 3 个函数都返回 `false` 时, 函数 `good()` 返回 `true`。

### 27.9.3 打开文件进行输入和输出

要使用文件, 必须首先打开它。要使用 `ofstream` 打开文件 `myfile.cpp`, 声明一个 `ofstream` 实例, 并将该文件名作为参数传递给它:



```
ofstream fout("myfile.cpp");
```

上述代码打开该文件以便进行输出。打开该文件以便进行输入的方法完全相同，只是需要使用 `ifstream` 对象：

```
ifstream fin("myfile.cpp");
```

`fout` 和 `fin` 是您指定的名称。这里使用 `fout` 旨在说明它与 `cout` 相似，而使用 `fin` 旨在说明它与 `cin` 相似。也可以使用指出它们要访问的文件的名称。

稍后就将使用的一个重要文件流函数是 `close()`。您创建的每个文件流对象都打开一个文件，以便进行读、写或读写。完成读写后应关闭文件，这非常重要，它确保文件不会受损以及将缓冲区的数据写入到磁盘。

将流对象同文件关联起来后，可以像使用其他流对象一样使用它们。程序清单 27.16 演示了这一点。

程序清单 27.16 打开文件以便进行读写

```
1: #include <fstream>
2: #include <iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     char fileName[80];
8:     char buffer[255];    // for user input
9:     cout << "File name: ";
10:    cin >> fileName;
11:
12:    ofstream fout(fileName); // open for writing
13:    fout << "This line written directly to the file...\n";
14:    cout << "Enter text for the file: ";
15:    cin.ignore(1, '\n'); // eat the newline after the file name
16:    cin.getline(buffer, 255); // get the user's input
17:    fout << buffer << "\n"; // and write it to the file
18:    fout.close();           // close the file, ready for reopen
19:
20:    ifstream fin(fileName); // reopen for reading
21:    cout << "Here's the contents of the file:\n";
22:    char ch;
23:    while (fin.get(ch))
24:        cout << ch;
25:
26:    cout << "\n***End of file contents.***\n";
27:
28:    fin.close();           // always pays to be tidy
29:    return 0;
30: }
```

#### ▼ 输出：

```
File name: test1
Enter text for the file: This text is written to the file!
Here's the contents of the file:
This line written directly to the file...
This text is written to the file!

***End of file contents.***
```

#### ▼ 分析：

第 7 行创建了一个用于存储文件名的字符数组。第 8 行创建了一个用于存储用户输入的字符数组。第 9 行提示用户输入一个文件名，输入被存储到数组 `fileName` 中。第 12 行创建一个名为 `fout` 的 `ofstream` 对象，并将其与用户输入的文件名关联起来。这将打开该文件，如果该文件已经存在，将删除其内容。

第 13 行将一个文本字符串直接写入文件。第 14 行提示用户输入文件内容。第 15 行使用函数 `ignore()` 删除用户输入文件名时留下的换行符。第 16 行将用户输入的文件内容存储到数组 `buffer` 中。第 17 行

将用户的输入和一个换行符写入文件。然后第 18 行关闭文件。

第 20 行重新打开该文件，但这次使用 `ifstream` 对象以输入模式打开。然后第 23 行和第 24 行以每次一个字符的方式读取文件的内容。

## 27.9.4 修改 `ofstream` 打开文件时的默认行为

打开文件时的默认行为是，如果文件不存在则创建它，如果文件已经存在则删除其内容。如果不想采用打开文件时的默认行为，可以给 `ofstream` 类的构造函数提供第 2 个参数。

该参数的合法取值有以下 5 项。

- `ios::app`：附加到已有文件的末尾，而不是删除其内容。
- `ios::ate`：跳到文件末尾，但可以在文件的任何地方写入数据。
- `ios::trunc`：默认值。删除已有文件的内容。
- `ios::nocreate`：如果文件不存在，打开操作失败。
- `ios::nocreplace`：如果文件已经存在，打开操作失败。

注意，`app` 是 `append` 的缩写；`ate` 是 `at end` 的缩写，而 `trunc` 是 `truncate` 的缩写。程序清单 27.17 重新打开程序清单 27.16 使用的文件并追加内容，演示了追加模式的使用法。

程序清单 27.17 在文件末尾添加内容

```

1: #include <fstream>
2: #include <iostream>
3: using namespace std;
4:
5: int main()    // returns 1 on error
6: {
7:     char fileName[80];
8:     char buffer[255];
9:     cout << "Please reenter the file name: ";
10:    cin >> fileName;
11:
12:    ifstream fin(fileName);
13:    if (fin)          // already exists?
14:    {
15:        cout << "Current file contents:\n";
16:        char ch;
17:        while (fin.get(ch))
18:            cout << ch;
19:        cout << "\n***End of file contents.***\n";
20:    }
21:    fin.close();
22:
23:    cout << "\nOpening " << fileName << " in append mode...\n";
24:
25:    ofstream fout(fileName,ios::app);
26:    if (!fout)
27:    {
28:        cout << "Unable to open " << fileName << " for appending.\n";
29:        return(1);
30:    }
31:
32:    cout << "\nEnter text for the file: ";
33:    cin.ignore(1,'\n');
34:    cin.getline(buffer,255);
35:    fout << buffer << "\n";
36:    fout.close();
37:
38:    fin.open(fileName); // reassign existing fin object!
39:    if (!fin)
40:    {
41:        cout << "Unable to open " << fileName << " for reading.\n";
42:        return(1);
43:    }

```

```

44:     cout << "\nHere's the contents of the file:\n";
45:     char ch;
46:     while (fin.get(ch))
47:         cout << ch;
48:     cout << "\n***End of file contents.***\n";
49:     fin.close();
50:     return 0;
51: }

```

### ▼ 输出:

```

Please reenter the file name: test1
Current file contents:
This line written directly to the file...
This text is written to the file!

***End of file contents.***

Opening test1 in append mode...

Enter text for the file: More text for the file!

Here's the contents of the file:
This line written directly to the file...
This text is written to the file!
More text for the file!

***End of file contents.***

```

### ▼ 分析:

和前一个程序清单一样,第 9 行和第 10 行也提示用户输入文件名,但第 12 行创建了一个输入文件流对象。第 13 行对打开操作进行测试,如果文件已经存在,则第 15~19 行打印其内容。注意,if(fin) 和 if(four.good()) 等效。

然后关闭该文件,第 25 行重新打开该文件,但使用追加模式。在这次(以及每次)打开文件后,都对文件进行测试以核实它被正确打开。注意,if(!fout) 和测试 if(fout.fail()) 等效。如果文件没有打开,第 28 行打印一条错误消息,然后使用一条 return 语句结束程序。如果文件被成功打开,则提供用户输入文件内容,然后第 36 行再次关闭文件。

最后,和程序清单 27.16 一样,以读取模式再次打开文件,但这次不必再次声明 fin,而只是将相同的文件与其关联起来。同样,第 39 行对文件打开操作进行测试,如果一切正常,将文件内容打印到屏幕上,然后关闭文件。

#### 应该

每次打开文件时都应进行测试以核实文件被成功打开。  
应重用已有的 ifstream 和 ofstream 对象。  
使用完 fstream 对象后应关闭它们。

#### 不应该

不要企图关闭 cin 和 cout 或给它们重新赋值。  
尽可能不要在 C++ 程序中使用 printf()。

## 27.10 二进制文件和文本文件

有些操作系统(如 DOS)区分二进制文件和文本文件。文本文件将所有内容都保存为文本(读者可能猜到了),因此像 54 325 这样的大数被保存为一串数字('5'、'4'、'3'、'2'、'5')。这样做效率很低,但优点是可以利用诸如 DOS 命令和 Windows 命令程序 type 等简单程序来阅读文本。

为帮助文件系统区分二进制文件和文本文件,C++ 提供了标记 ios::binary。在很多系统上该标记都被忽略,因为所有数据都以二进制方式存储。在一些非常严谨的系统上,ios::binary 标记是非法的,甚至不能通过编译。

二进制文件不仅能够存储整数和字符串，还能够存储整个数据结构。可以使用 `fstream` 的 `write()` 方法一次性写入所有的数据。

如果使用 `write()` 写入，可以用 `read()` 来读取。然而，这两个函数都接受一个字符指针作为参数，因此必须将类的地址强制转换为字符指针。

这些函数的第二个参数是要读写的字符数，这可以使用 `sizeof()` 来确定。注意，写入的只是类的数，而不包括方法；读取的也只有数据。程序清单 27.18 演示了如何将对象的内容写入文件。

程序清单 27.18 将对象的内容写入文件

```

1: #include <fstream>
2: #include <iostream>
3: using namespace std;
4:
5: class Animal
6: {
7:     public:
8:         Animal(int weight,long days):itsWeight(weight),DaysAlive(days){}
9:         ~Animal(){}
10:
11:         int GetWeight()const { return itsWeight; }
12:         void SetWeight(int weight) { itsWeight = weight; }
13:
14:         long GetDaysAlive()const { return DaysAlive; }
15:         void SetDaysAlive(long days) { DaysAlive = days; }
16:
17:     private:
18:         int itsWeight;
19:         long DaysAlive;
20: };
21:
22: int main()    // returns 1 on error
23: {
24:     char fileName[80];
25:
26:
27:     cout << "Please enter the file name: ";
28:     cin >> fileName;
29:     ofstream fout(fileName,ios::binary);
30:     if (!fout)
31:     {
32:         cout << "Unable to open " << fileName << " for writing.\n";
33:         return(1);
34:     }
35:
36:     Animal Bear(50,100);
37:     fout.write((char*) &Bear,sizeof Bear);
38:
39:     fout.close();
40:
41:     ifstream fin(fileName,ios::binary);
42:     if (!fin)
43:     {
44:         cout << "Unable to open " << fileName << " for reading.\n";
45:         return(1);
46:     }
47:
48:     Animal BearTwo(1,1);
49:
50:     cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
51:     cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
52:
53:     fin.read((char*) &BearTwo, sizeof BearTwo);
54:
55:     cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
56:     cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
57:     fin.close();
58:     return 0;
59: }
```

**▼ 输出:**

```
Please enter the file name: Animals
BearTwo weight: 1
BearTwo days: 1
BearTwo weight: 50
BearTwo days: 100
```

**▼ 分析:**

第 5~20 行声明了一个简化的 `Animal` 类。第 24~34 行创建了一个文件，并以二进制模式打开它进行输出。第 36 行创建了一个体重为 50、年龄是 100 天的 `Animal` 对象，第 37 行将其数据写入文件。

第 39 行关闭该文件，第 41 行以二进制模式重新打开该文件以便进行读取。第 48 行创建了第二个 `Animal` 对象，其体重为 1，年龄只有 1 天。第 53 行将文件中的数据读入新 `Animal` 对象中，这将用文件中的数据替换原来的数据，输出证明了这一点。

## 27.11 命令行处理

很多操作系统（如 DOS 和 Unix）都允许用户在启动程序时给它传递参数。这些参数被称为命令行选项，通常在命令行上用空格将它们分开。例如：

```
SomeProgram Param1 Param2 Param3
```

这些参数并不直接传递给 `main()` 函数。相反，每个程序的 `main()` 函数都被传入两个参数。第一个参数是一个整数，它指定了命令行参数数目，其中包括程序名本身，因此每个程序至少有一个参数。前面给出的命令行示例包含 4 个参数（程序名 `SomeProgram` 和 3 个参数，这样总共是 4 个命令行参数）。

传递给函数 `main()` 的第 2 个参数是一个字符串指针数组。由于数组名是一个指向数组第 1 个元素的常量指针，因此可以将参数声明为指向字符指针的指针、指向字符数组的指针或字符数组的数组。

第 1 个参数通常名为 `argc`（参数数目），但也可以给它指定其他名称；第 2 个参数通常名为 `argv`（参数向量），但这只是一种约定。

通常应检测 `argc` 以核实 `main()` 函数接受到预期的参数数目，并使用 `argv` 来访问字符串本身。注意 `argv[0]` 是程序名，`argv[1]` 是以字符串表示的程序的第一个参数。如果程序接受两个数字作为参数，需要将这些数字转换为字符串。程序清单 27.19 演示了如何使用命令行参数。

程序清单 27.19 使用命令行参数

```
1: #include <iostream>
2: int main(int argc, char **argv)
3: {
4:     std::cout << "Received " << argc << " arguments...\n";
5:     for (int i=0; i<argc; i++)
6:         std::cout << "argument " << i << ": " << argv[i] << std::endl;
7:     return 0;
8: }
```

**▼ 输出:**

```
TestProgram Teach Yourself C++ In 21 Days
Received 7 arguments...
argument 0: TestProgram
argument 1: Teach
argument 2: Yourself
argument 3: C++
argument 4: In
argument 5: 21
argument 6: Days
```

**注意**

必须在命令行（也就是 DOS 窗口中）运行上述代码，或者在编译器中设置命令行参数（请参阅编译器文档）。

## ▼ 分析:

函数 `main()` 接受 2 个参数: `argc` 是一个表示命令行参数个数的整数, `argv` 是一个指向字符串数组的指针。在 `argv` 指向的数组中, 每个字符串都是一个命令行参数。注意, 也可以将 `argv` 声明为 `char *argv[]` 或 `char argv[][]`。如何声明 `argv` 是一个编程风格问题, 虽然该程序将它声明为指向指针的指针, 但仍可以使用下标表示法来访问各个字符串。

第 4 行使用 `argc` 来打印命令行参数个数: 总共有 7 个, 其中包括程序名本身。

第 5 行和第 6 行打印每个命令行参数: 使用下标表示法将以空字符结尾的字符串传递给 `cout`。

程序清单 27.20 演示了一种更常见的命令行参数的用途, 它修改了程序清单 27.18, 以命令行参数的方式来获取文件名。

程序清单 27.20 使用命令行参数来获取文件名

```
1: #include <fstream>
2: #include <iostream>
3: using namespace std;
4:
5: class Animal
6: {
7:     public:
8:         Animal(int weight, long days): itsWeight(weight), DaysAlive(days){}
9:         ~Animal(){}
10:
11:         int GetWeight()const { return itsWeight; }
12:         void SetWeight(int weight) { itsWeight = weight; }
13:
14:         long GetDaysAlive()const { return DaysAlive; }
15:         void SetDaysAlive(long days) { DaysAlive = days; }
16:
17:     private:
18:         int itsWeight;
19:         long DaysAlive;
20: };
21:
22: int main(int argc, char *argv[]) // returns 1 on error
23: {
24:     if (argc != 2)
25:     {
26:         cout << "Usage: " << argv[0] << " <filename>" << endl;
27:         return(1);
28:     }
29:
30:     ofstream fout(argv[1], ios::binary);
31:     if (!fout)
32:     {
33:         cout << "Unable to open " << argv[1] << " for writing.\n";
34:         return(1);
35:     }
36:
37:     Animal Bear(50, 100);
38:     fout.write((char*) &Bear, sizeof Bear);
39:
40:     fout.close();
41:
42:     ifstream fin(argv[1], ios::binary);
43:     if (!fin)
44:     {
45:         cout << "Unable to open " << argv[1] << " for reading.\n";
46:         return(1);
47:     }
48:
49:     Animal BearTwo(1, 1);
50:
51:     cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
52:     cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
53:
54:     fin.read((char*) &BearTwo, sizeof BearTwo);
```



```
55:
56:     cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
57:     cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
58:     fin.close();
59:     return 0;
60: }
```

#### ▼ 输出:

```
BearTwo weight: 1
BearTwo days: 1
BearTwo weight: 50
BearTwo days: 100
```

#### ▼ 分析:

Animal 类的声明和程序清单 27.18 完全相同,但这次没有提示用户输入文件名,而是使用命令行参数来获取。第 22 行将 main() 函数声明为接受 2 个参数:命令行参数个数和指向命令行参数字符串数组的指针。

在第 24~28 行,程序确保收到预期的参数个数(2 个)。如果用户没有提供一个文件名,则打印一条错误消息:

```
Usage TestProgram <filename>
```

然后程序退出。注意,通过使用 argv[0] 而不是程序名,可以把将该程序编译为任何名称,而不会妨碍该语句的执行结果。甚至可以将编译后的可执行文件重命名,该语句仍将正确执行。

在第 30 行,程序试图以二进制输出模式打开指定的文件。没有理由将文件名复制到局部临时变量中,可以通过访问 argv[1] 来直接使用它。

第 42 行再次使用了这种技巧,重新打开该文件进行输入。当文件不能打开时,错误状态语句也使用了这种技巧(第 33 行和第 45 行)。

## 27.12 总结

本章介绍了流,讨论了全局对象 cin 和 cout。istream 和 ostream 对象旨在封装写入设备驱动程序以及缓冲输入和输出的工作。

在每个程序中都创建了 4 个标准流对象:cout、cin、cerr 和 clog。在很多操作系统中,这些对象都可以被重定向。

istream 对象 cin 用于输入,它通常与重载的提取运算符(>>)一起使用;ostream 对象 cout 用于输出,它通常与重载的插入运算符(<<)一起使用。

这些对象还包含很多其他的成员函数,如 get() 和 put()。由于这些成员函数的常用版本返回一个流对象引用,因此可以级联这些运算符和函数。

可以使用控制符来修改流对象的状态。它们可以设置流对象的格式化和显示特征以及其他各种属性。

可以使用从流类派生而来的 fstream 类来实现文件 I/O。除支持插入和提取运算符外,这些对象还支持用于存储和检索大型二进制对象的函数 read() 和 write()。

## 27.13 问与答

问:如何确定何时使用插入和提取运算符,何时使用流类的其他成员函数?

答:一般而言,使用插入和提取运算符较容易,在其行为能够满足需求时应首选它们。在这些运算符不能满足需求的非常情况下(如读取包含多个单词的字符串),可使用其成员函数。

问:cerr 和 clog 之间的区别是什么?

答: `cerr` 不被缓冲, 写入 `cerr` 的内容立即被输出。需要将错误消息写入屏幕时, 这很合适, 但将日志写入磁盘时, 这将极大地影响程序的性能。`Clog` 对输出进行缓冲, 因此效率更高, 但风险是如果程序崩溃将丢失部分日志数据。

问: 既然 `printf()` 很管用为什么还要创建流?

答: `printf()` 不支持 C++ 的强类型系统, 也不支持用户定义的类。对 `printf()` 的支持实际上是对 C 编程语言的传承。

问: 什么时候使用 `putback()`?

答: 在使用读操作来确定字符是否有效, 但另一个读操作 (可能由另一个对象执行) 需要该字符位于缓冲区中时。这种情况大都发生在分析文件时, 例如, C++ 编译器可能使用 `putback()`。

问: 我的朋友在其 C++ 程序中使用 `printf()`, 我也可以这样做吗?

答: 不可以。现在应将 `printf()` 视为被摒弃的。

## 27.14 作业

作业包括测验和练习, 前者帮助加深读者对所学知识的理解, 后者提供了使用新学知识的机会。请尽量先完成测验和练习题, 然后再对照附录 D 的答案, 继续学习下一章前, 请务必弄懂这些答案。

### 27.14.1 测验

1. 什么是插入运算符? 其作用是什么?
2. 什么是提取运算符? 其作用是什么?
3. `cin.get()` 的 3 种形式是什么? 它们之间有什么区别?
4. `cin.read()` 和 `cin.getline()` 之间有什么不同?
5. 使用插入运算符输出 `long` 时默认宽度是多少?
6. 插入运算符的返回值是什么?
7. `ofstream` 的构造函数接受什么参数?
8. 参数 `ios::ate` 有何作用?

### 27.14.2 练习

1. 编写一个程序, 将数据写入 4 个标准 `iostream` 对象: `cin`、`cout`、`cerr` 和 `clog`。
2. 编写一个程序, 提示用户输入其全名, 然后将其显示在屏幕上。
3. 重写程序清单 27.9, 使其功能保持不变, 但不使用 `putback()` 和 `ignore()`。
4. 编写一个程序, 它接受一个文件名作为参数, 并打开这个文件进行读取。读取该文件的每个字符, 但只将字母和标点符号显示到屏幕上 (忽略所有的非打印字符), 然后关闭文件并退出。
5. 编写一个程序, 按相反的顺序显示其命令行参数, 但不显示程序名。

# 第 28 章

## 处理异常

本书前面的程序都是为演示而创建的，它们没有处理错误以便读者能够将重点放在重要主题上。实际的程序必须考虑错误状态。

在本章中，您将学习：

- 什么是异常
- 如何使用异常？它们会带来什么问题
- 如何建立异常层次结构
- 如何使异常符合整体错误处理方法
- 什么是调试器

### 28.1 程序中的各种错误

实际的程序很少没有 bug。程序越大，有 bug 的可能越大。实际上，很多 bug “走出家门” 进入了最终发布的软件中。编写健壮、没有 bug 的程序是任何对编程持严肃态度者的首要任务。

软件业中的最大问题就是有错误、不稳定的代码。在很多重要的编程工作中，花费最大的是测试和修改。以低成本按时生产出优秀、稳定、可靠的程序的人将给软件业带来革命性影响。

各种 bug 将给程序带来麻烦。首先是逻辑性差：程序能够完成要求的工作，但您并没有正确地考虑算法。其次是语法问题：使用了错误的惯用语、函数或结构。这两种是最常见的，它们是大多数程序员都要警惕的错误。

研究和实际经验表明，在开发过程中发现逻辑问题的时间越晚，修复它所需付出的代价越高。费用最低的修复问题的方法是尽量避免产生错误，其次是编译器可发现的错误。C++标准要求编译器尽可能让更多 bug 在编译时浮出水面。

与只是偶尔导致程序崩溃的错误相比，发现和修复让程序能够通过编译但首次测试时就会出现错误（导致程序每次运行时都崩溃的错误）的代价更低些。

比逻辑和语法错误更常见的运行阶段问题是脆弱性：程序要求输入一个数字时，如果用户输入一个数字，程序将正常运行；但如果用户输入一个字母，程序将崩溃。另一些程序在内存不足、软盘不在软驱中或 Internet 连接断开时崩溃。

为避免这种脆弱性，程序员致力于提高程序的健壮性。健壮 (bulletproof) 的程序能够应付运行阶段发生的任何问题，从怪异用户输入到内存不足。

区分 bug、逻辑错误和异常至关重要。bug 是由于程序员犯错引起的，逻辑错误是由于程序员对解决问题的方式不了解引起的；异常是由于不常见、但可预见的问题（如内存或硬盘空间等资源耗尽）引起的。

### 异常情况

您无法消除异常情况，只能为异常情况做好准备。如果程序要动态地为对象分配内存，但系统没

有内存可用，将发生什么？程序将如何应对？如果您导致常见的数学错误之一（被零除），程序将如何做？程序的选择包括：

- 崩溃；
- 通知用户并妥善地退出；
- 通知用户并允许用户尽量恢复并继续执行；
- 采取正确的措施，在不影响用户的情况下继续运行。

请看程序清单 28.1，它非常简单且运行时将崩溃，但演示了很多程序都可能出现的极其严重的问题。

程序清单 28.1 导致异常情形

```
0: // This program will crash
1: #include <iostream>
2: using namespace std;
3:
4: const int DefaultSize = 10;
5:
6: int main()
7: {
8:     int top = 90;
9:     int bottom = 0;
10:
11:     cout << "top / 2 = " << (top / 2) << endl;
12:
13:     cout << "top divided by bottom = ";
14:     cout << (top / bottom) << endl;
15:
16:     cout << "top / 3 = " << (top / 3) << endl;
17:
18:     cout << "Done." << endl;
19:     return 0;
20: }
```

#### ▼ 输出：

```
top / 2 = 45
top divided by bottom =
```

#### 警告

该程序可能在控制台显示上述输出，但随后很可能立刻崩溃。

#### ▼ 分析：

程序清单 28.1 有意被设计成导致崩溃。然而，如果程序要求用户输入两个数，并将它们相除，也可能出现这样的问题。

第 8 行和第 9 行声明了两个 int 变量并给它们赋值。也可以提示用户输入两个数字或从文件中读取。在第 11、14 和 16 行，这些数字被用于数学运算。具体地说，它们被用于除法运算。在第 11 行和第 16 行，没有任何问题，但第 14 行存在严重的问题。除以零将导致异常问题发生：程序崩溃。程序将终止，操作系统很可能会显示一条异常消息。

虽然并非总是必须（乃至希望）悄悄地自动地从所有异常情形恢复，但显然必须做得比这个程序更好些：不能让程序崩溃。

C++ 的异常处理提供了一种类型安全的集成方法，来应对程序运行时出现的可预见到但不常发生的情形。

## 28.2 异常的基本思想

异常的基本思想非常简单。

- 计算机试图执行一段代码。这段代码可能要分配资源（如内存）、锁定文件或执行其他各种任务。
- 包含应对代码由于异常原因而执行失败的逻辑（代码）。例如，可能包含捕获各种问题（如无

法分配内存、无法锁定文件或各种其他的问题)的代码。

- 在您的代码被其他代码使用时(如一个函数调用另一个),也需要一种机制来将有关问题(异常)的信息传递到下一级。应该有一条从问题发生的代码到处理错误状态的代码的路径。如果函数之间存在中间层,应该给它们提供解决问题的机会,但不应要求它们包含只是为了传递错误状态的代码。

异常处理将这3点结合在一起,且工作原理相对比较简单。

## 28.2.1 异常处理的组成部分

要处理异常,必须首先确定需要监视哪段代码可能发生的异常。这可以使用 try 块来完成。

应创建 try 块来包围可能导致问题的代码块。try 块的基本格式如下:

```
try
{
    SomeDangerousFunction();
}
catch (...)
{
}
```

在这个例子中,当 SomeDangerousFunction() 执行时,如果发生任何异常,将被发现并捕获。要监视异常,只需加上关键字 try 和大括号。当然,如果异常发生,需要采取措施来处理它。

当 try 块中的代码执行时,如果发生异常,则被称为引发异常。然后可以捕获引发的异常,如前一个示例所示,使用 catch 块来捕获异常。引发异常后,将转移到当前 try 块后面合适的 catch 块执行。在前一个示例中,省略号 (...) 表示任何异常。也可以捕获特定类型的异常,为此,可以在 try 块后面使用一个或多个 catch 块。例如:

```
try
{
    SomeDangerousFunction();
}
catch(OutOfMemory)
{
    // take some actions
}
catch(FileNotFound)
{
    // take other action
}
catch (...)
{
}
```

在这个例子中,当 SomeDangerousFunction() 被执行时,有处理异常的代码。如果引发了异常,它将被传递给 try 块后面的第一个 catch 块。如果该 catch 块有类型参数(就像前一个示例中那样),将检查异常是否与指定的类型匹配。如果不匹配,将检查下一条 catch 语句,这一过程不断进行下去,直到找到匹配的 catch 块或到达非 catch 块代码。找到第一个匹配的 catch 块后将执行它。除非有意想让某些类型的异常逃脱,否则总是应该让最后一个 catch 块使用省略号作为参数。

**注意**

catch 块也被称为处理程序,因为它能够处理异常。

**注意**

可以将 catch 块视为类似于被重载的函数,找到特征标匹配的函数后,将执行它。

处理异常的基本步骤如下:

1. 确定程序中执行某种操作且可能引发异常的代码,并将它们放到 try 块中;
2. 创建 catch 块,在异常被引发时捕获它们。可以创建捕获特定类型的异常的 catch 块(通过指定 catch 块的类型参数),也可以创建捕获所有异常的 catch 块(使用省略号作为参数)。

程序清单 28.2 在程序清单 28.1 中添加了基本的异常处理功能,这里使用了 try 块和 catch 块。



**注意**

有些非常老的编译器不支持异常。然而，异常是 ANSI C++ 标准的一部分，每个编译器厂商的最新编译器版本都全面支持异常。如果读者使用的是较老的编译器，将无法编译和运行本章的示例。然而，通读本章并在对编译器进行升级后复习这些内容，仍不失为一个不错的主意。

**程序清单 28.2 捕获异常**

```

0: // trying and catching
1: #include <iostream>
2: using namespace std;
3:
4: const int DefaultSize = 10;
5:
6: int main()
7: {
8:     int top = 90;
9:     int bottom = 0;
10:
11:     try
12:     {
13:         cout << "top / 2 = " << (top/ 2) << endl;
14:
15:         cout << "top divided by bottom = ";
16:         cout << (top / bottom) << endl;
17:
18:         cout << "top / 3 = " << (top/ 3) << endl;
19:     }
20:     catch(...)
21:     {
22:         cout << "something has gone wrong!" << endl;
23:     }
24:
25:     cout << "Done." << endl;
26:     return 0;
27: }
```

**▼ 输出:**

```

top / 2 = 45
top divided by bottom = something has gone wrong!
Done.
```

**▼ 分析:**

不同于前一个程序清单，执行程序清单 28.2 不会导致崩溃，它能够报告问题并妥善地退出。

这次将可能发生问题的代码（除法运算）放在 try 块（第 11~19 行）中。为处理异常，try 块后面包含一个 catch 块（第 20~23 行）。

第 20 行的 catch 语句中包含一个省略号。正如前面指出的，这是一种特殊的 catch 语句。除非异常被之前的 catch 块处理了，否则前面的 try 块中的代码导致的所有异常都将由该 catch 块处理。在该程序清单中，只可能发生被零除的错误。正如后面将介绍的，最好考虑更具体的异常类型，这样可以自定义每种异常的处理方式。

读者应该注意到了，该程序清单执行时没有崩溃。另外，从输出可知，程序继续执行了 catch 语句后面的第 25 行，单词 Done 被打印到控制台证明了这一点。

**try 块**

try 块是以关键字 try 和 { 打头且以 } 结尾的一系列语句。

**示例:**

```

try
{
    Function();
}
```



**catch 块**

catch 块是以关键字 catch、用括号括起的异常类型和{打头且以}结束的一系列语句。

示例:

```
try
{
    Function();
}
catch (OutOfMemory)
{
    // take action
}
```

## 28.2.2 手工引发异常

程序清单 28.2 演示了异常处理的两个方面: 标记要监控的代码和指定要如何处理异常, 然而只处理预定义的异常。异常处理的第 3 部分是, 让您能够创建自己的、将被处理的异常类型。通常创建自己的异常, 可以自定义异常处理程序 (catch 块), 使其对您的应用程序是有意义的。

要创建导致 try 语句对其做出反应的异常, 可使用关键字 throw。实际上, 您引发异常, 而处理程序 (catch 块) 可能捕获它。throw 语句的基本格式如下:

```
throw exception;
```

该语句引发异常 exception。这将导致程序跳到一个处理程序处执行, 如果没有找到匹配的处理程序, 程序将终止。

引发异常时, 几乎可以使用任何类型的值。正如前面指出的, 可以为程序可能引发的每种异常编写相应的处理程序。程序清单 28.3 对程序清单 28.2 进行了修改, 演示了如何引发一种基本异常。

程序清单 28.3 引发异常

```
0: //Throwing
1: #include <iostream>
2:
3: using namespace std;
4:
5: const int DefaultSize = 10;
6:
7: int main()
8: {
9:     int top = 90;
10:    int bottom = 0;
11:
12:    try
13:    {
14:        cout << "top / 2 = " << (top/ 2) << endl;
15:
16:        cout << "top divided by bottom = ";
17:        if ( bottom == 0 )
18:            throw "Division by zero!";
19:
20:        cout << (top / bottom) << endl;
21:
22:        cout << "top / 3 = " << (top/ 3) << endl;
23:    }
24:    catch( const char * ex )
25:    {
26:        cout << "\n*** " << ex << " ***" << endl;
27:    }
28:
29:    cout << "Done." << endl;
30:    return 0;
31: }
```

## ▼ 输出:

```
top / 2 = 45
top divided by bottom = *** Division by zero! ***
Done.
```

## ▼ 分析:

不同于前一个程序清单, 该程序清单更严格地控制了其异常。虽然这不是使用异常的最佳方式, 但清楚地演示了如何使用 throw 语句。

第 17 行检查 bottom 的值是否为零。如果是则引发异常。在这个例子中, 异常是一个字符串值。

处理程序从第 24 行的 catch 语句开始。该程序程序捕获常量字符指针。在异常方面, 字符串与常量字符指针匹配, 因此从第 24 行开始的处理程序将捕获第 18 行引发的异常。第 26 行显示传递的字符串, 并在前面和后面加上星号。第 27 行的右大括号表明处理程序到此结束, 因此跳到 catch 语句后面的第一行, 并继续执行到程序末尾。

如果异常是一个更严重的问题, 可以在第 26 行打印消息后退出应用程序。如果在函数中引发异常, 而该函数被其他函数调用, 可以将异常向上传递。要将异常向上传递, 只需调用 throw 命令且不提供任何参数, 这将在当前位置重新引发现有的异常。

### 28.2.3 创建异常类

可以创建一个复杂得多的类供引发异常时使用。程序清单 28.4 是一个精简的 Array 类, 它是一种非常原始的动态数组的实现。

程序清单 28.4 引发异常

```
0: #include <iostream>
1: using namespace std;
2:
3: const int DefaultSize = 10;
4:
5: class Array
6: {
7:     public:
8:         // constructors
9:         Array(int itsSize = DefaultSize);
10:        Array(const Array &rhs);
11:        ~Array() { delete [] pType;}
12:
13:        // operators
14:        Array& operator=(const Array&);
15:        int& operator[](int offSet);
16:        const int& operator[](int offSet) const;
17:
18:        // accessors
19:        int GetitsSize() const { return itsSize; }
20:
21:        // friend function
22:        friend ostream& operator<< (ostream&, const Array&);
23:
24:        class xBoundary {}; // define the exception class
25:
26:    private:
27:        int *pType;
28:        int itsSize;
29: };
30:
31: Array::Array(int size):
32:     itsSize(size)
33: {
```

```

34:     pType = new int[size];
35:     for (int i = 0; i < size; i++)
36:         pType[i] = 0;
37: }
38:
39: Array& Array::operator=(const Array &rhs)
40: {
41:     if (this == &rhs)
42:         return *this;
43:     delete [] pType;
44:     itsSize = rhs.GetitsSize();
45:     pType = new int[itsSize];
46:     for (int i = 0; i < itsSize; i++)
47:     {
48:         pType[i] = rhs[i];
49:     }
50:     return *this;
51: }
52:
53: Array::Array(const Array &rhs)
54: {
55:     itsSize = rhs.GetitsSize();
56:     pType = new int[itsSize];
57:     for (int i = 0; i < itsSize; i++)
58:     {
59:         pType[i] = rhs[i];
60:     }
61: }
62:
63: int& Array::operator[](int offSet)
64: {
65:     int size = GetitsSize();
66:     if (offSet >= 0 && offSet < GetitsSize())
67:         return pType[offSet];
68:     throw xBoundary();
69:     return pType[0]; // appease MSC
70: }
71:
72: const int& Array::operator[](int offSet) const
73: {
74:     int mysize = GetitsSize();
75:     if (offSet >= 0 && offSet < GetitsSize())
76:         return pType[offSet];
77:     throw xBoundary();
78:     return pType[0]; // appease MSC
79: }
80:
81: ostream& operator<< (ostream& output, const Array& theArray)
82: {
83:     for (int i = 0; i < theArray.GetitsSize(); i++)
84:         output << "[" << i << "]" << theArray[i] << endl;
85:     return output;
86: }
87:
88: int main()
89: {
90:     Array intArray(20);
91:     try
92:     {
93:         for (int j = 0; j < 100; j++)
94:         {
95:             intArray[j] = j;
96:             cout << "intArray[" << j << "] okay..." << endl;
97:         }
98:     }
99:     catch (Array::xBoundary)
100:    {
101:        cout << "Unable to process your input!" << endl;
102:    }
103:    cout << "Done." << endl;
104:    return 0;
105: }

```

## ▼ 输出:

```
intArray[0] okay...
intArray[1] okay...
intArray[2] okay...
intArray[3] okay...
intArray[4] okay...
intArray[5] okay...
intArray[6] okay...
intArray[7] okay...
intArray[8] okay...
intArray[9] okay...
intArray[10] okay...
intArray[11] okay...
intArray[12] okay...
intArray[13] okay...
intArray[14] okay...
intArray[15] okay...
intArray[16] okay...
intArray[17] okay...
intArray[18] okay...
intArray[19] okay...
Unable to process your input!
Done.
```

## ▼ 分析:

程序清单 28.4 使用了一个精简的 Array 类，但添加了异常处理功能，以防超出边界。

第 24 行为类 Array 的声明中声明了一个新类 xBoundary。无法看出这个新类是异常类，它和其他类没有任何区别。这个类极为简单，它没有数据和方法，但确实是一个有效的类。实际上，说它没有方法是不对的，因为编译器将自动为它创建一个默认构造函数、析构函数、复制构造函数和赋值运算符。因此它实际上有 4 个成员函数，但没有数据。注意，在 Array 中声明它只是为了将它们连接起来。Array 在访问 xBoundary 方面没有特权，xBoundary 在访问 Array 的成员方面也没有特权。

在第 63~70 行以及第 72~79 行，对下标运算符进行了修改，使之对下标进行检查。如果下标不在有效范围内，则将 xBoundary 类作为异常进行引发。括号是必不可少的，这样才是调用 xBoundary 的构造函数，而不是使用一个枚举常量。

在 main() 函数中，第 90 行声明了一个能够存储 20 个元素的 Array 对象。第 91 行的关键字 try 表明接下来是一个 try 块，该 try 块到第 98 行结束。在 try 块中，将 101 个整数添加到第 90 行声明的数组中。

在第 99 行，处理程序被声明为捕获 xBoundary 异常。

在第 88~105 行的 main() 函数中，创建了一个 try 块，并在其中对数组的每个成员进行初始化。当 j (第 93 行) 递增到 20 时，将访问下标为 20 的成员。这导致第 66 行的测试失败，operatorp[] 中的第 67 行代码引发 xBoundary 异常。

程序跳到第 99 行的 catch 块，该行的 catch 捕获 (处理) 异常：打印一条错误消息。程序执行到 catch 块的末尾 (第 102 行)。

## 28.3 使用 try 块和 catch 块

确定在什么地方放置 try 块可能比较困难：可能引发异常的操作并非总是那么明显。下一个问题是在什么地方捕获异常。您也许想在分配内存的地方引发所有内存异常，但想在程序上层处理用户界面的地方捕获异常。

在确定 try 块的位置时，考虑在何处分配内存或资源。要监视的其他异常包括越过边界、无效输入等。至少应该在 main() 函数中所有代码的周围放置 try/catch 块。try/catch 块通常放在高级函数中，尤其是那些知道程序用户界面的函数中。例如，实用 (utility) 类通常不应捕获那些需要报告给用户的异常，因为这种类可能被用于窗口程序、控制台程序，甚至通过 Web 或消息收发功能与用户交流的程序中。

## 28.4 捕获异常的工作原理

捕获异常的工作原理如下：异常被引发后，将检查调用栈。调用栈是在程序的一部分调用另一个函数时创建的函数调用列表。

调用栈记录执行路径。如果 `main()` 调用了函数 `Animal::GetFavoriteFood()`，`GetFavoriteFood()` 调用了函数 `Animal::LookupPreferences()`，而后者又调用了 `fstream::operator>>()`，这些调用都将记录在调用栈中。递归函数可能在调用栈中出现多次。

异常沿调用栈向上传递给每个封闭块 (enclosing block)，这被称为堆栈解退 (unwinding the stack)。当堆栈被解退时，对堆栈中的局部对象调用析构函数，从而将对象销毁。

每个 `try` 块后面都有一个或多个 `catch` 语句。如果异常与某个 `catch` 语句匹配，将执行该 `catch` 语句并认为异常已得到处理。如果没有匹配的 `catch` 语句，将继续解退堆栈。如果异常直到程序开始位置 (`main()`) 还没有被捕获，将调用内置的处理程序来终止程序。

需要注意的是，异常沿堆栈向上传递是条单行线。在异常向上传递的过程中，堆栈被解退，堆栈中的对象被销毁。没有回头路可走：异常被捕获后，程序将继续执行捕获异常的 `catch` 语句后面的语句。

因此，在程序清单 28.4 中，程序继续执行第 101 行：捕获 `xBoundary` 异常的 `catch` 语句后面的第一行。引发异常后，程序跳到后面的 `catch` 块处继续执行，而不是继续执行异常引发点后面的代码。

### 28.4.1 使用多条 `catch` 语句

可能引发多种异常。在这种情况下，可以依次使用多条 `catch` 语句，就像 `switch` 语句中的条件一样。与 `default` 语句对应的是“捕获所有异常”的 `catch` 语句：`catch(...)`。程序清单 28.5 演示了多种异常。

程序清单 28.5 多种异常

```
0: #include <iostream>
1: using namespace std;
2:
3: const int DefaultSize = 10;
4:
5: class Array
6: {
7:     public:
8:         // constructors
9:         Array(int itsSize = DefaultSize);
10:        Array(const Array &rhs);
11:        ~Array() { delete [] pType; }
12:
13:        // operators
14:        Array& operator=(const Array&);
15:        int& operator[](int offSet);
16:        const int& operator[](int offSet) const;
17:
18:        // accessors
19:        int GetitsSize() const { return itsSize; }
20:
21:        // friend function
22:        friend ostream& operator<< (ostream&, const Array&);
23:
24:        // define the exception classes
25:        class xBoundary {};
26:        class xTooBig {};
27:        class xTooSmall {};
28:        class xZero {};
29:        class xNegative {};
30:    private:
31:        int *pType;
32:        int itsSize;
33: };
34:
```

```

35: int& Array::operator[](int offSet)
36: {
37:     int size = GetitsSize();
38:     if (offSet >= 0 && offSet < GetitsSize())
39:         return pType[offSet];
40:     throw xBoundary();
41:     return pType[0]; // appease MFC
42: }
43:
44:
45: const int& Array::operator[](int offSet) const
46: {
47:     int mysize = GetitsSize();
48:     if (offSet >= 0 && offSet < GetitsSize())
49:         return pType[offSet];
50:     throw xBoundary();
51:
52:     return pType[0]; // appease MFC
53: }
54:
55:
56: Array::Array(int size):
57:     itsSize(size)
58: {
59:     if (size == 0)
60:         throw xZero();
61:     if (size < 10)
62:         throw xTooSmall();
63:     if (size > 30000)
64:         throw xTooBig();
65:     if (size < 1)
66:         throw xNegative();
67:
68:     pType = new int[size];
69:     for (int i = 0; i < size; i++)
70:         pType[i] = 0;
71: }
72:
73: int main()
74: {
75:     try
76:     {
77:         Array intArray(0);
78:         for (int j = 0; j < 100; j++)
79:         {
80:             intArray[j] = j;
81:             cout << "intArray[" << j << "] okay..." << endl;
82:         }
83:     }
84:     catch (Array::xBoundary)
85:     {
86:         cout << "Unable to process your input!" << endl;
87:     }
88:     catch (Array::xTooBig)
89:     {
90:         cout << "This array is too big..." << endl;
91:     }
92:     catch (Array::xTooSmall)
93:     {
94:         cout << "This array is too small..." << endl;
95:     }
96:     catch (Array::xZero)
97:     {
98:         cout << "You asked for an array";
99:         cout << " of zero objects!" << endl;
100:     }
101:     catch (...)
102:     {
103:         cout << "Something went wrong!" << endl;
104:     }
105:     cout << "Done." << endl;
106:     return 0;
107: }

```



**▼ 输出:**

```
You asked for an array of zero objects!  
Done.
```

**▼ 分析:**

第 25~29 行声明了 4 个新类: xTooBig、xTooSmall、xZero 和 xNegative。第 56~71 行的构造函数检查传递给参数 size 的值。如果太大、太小、为零或负数,将引发异常。

在 try 块后面为每种异常提供了一条 catch 语句,但不包括异常 size 为负,这种异常将由第 101 行的“捕获所有异常”语句 catch(...)捕获。

请多次尝试将数组大小设置为不同的值。然后尝试输入-5,在这种情况下,读者可能认为将引发 xNegative 异常,然而构造函数中的测试顺序防止了这种情况发生:在判断 size<1 之前判断 size<10。要修复这种问题,可将第 61 行和第 62 行与第 65 行和第 66 行相换,然后重新编译。

**提示**

构造函数被调用后,便为对象分配了内存。因此,在构造函数中引发异常可能为对象分配了内存但该对象不可用。通常应该将构造函数放在 try/catch 块中,并在发生异常时将对象标记为不可用。每个成员函数都应检查该“有效”标记,以免使用初始化被中断的对象进而导致其他错误。

## 28.4.2 异常层次结构

异常是类,因此也可以从它们派生出其他类。也许创建一个 xSize 类,然后从它派生出 xZero、xTooSmall、xTooBig 和 xNegative 更合适。这样,有些函数可以只捕获 xSize 异常,而另一些函数可以捕获具体类型的 xSize 异常。程序清单 28.6 演示了这种想法。

程序清单 28.6 类层次结构和异常

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: const int DefaultSize = 10;  
4:  
5: class Array  
6: {  
7:     public:  
8:         // constructors  
9:         Array(int itsSize = DefaultSize);  
10:        Array(const Array &rhs);  
11:        ~Array() { delete [] pType; }  
12:  
13:        // operators  
14:        Array& operator=(const Array&);  
15:        int& operator[](int offSet);  
16:        const int& operator[](int offSet) const;  
17:  
18:        // accessors  
19:        int GetitsSize() const { return itsSize; }  
20:  
21:        // friend function  
22:        friend ostream& operator<< (ostream&, const Array&);  
23:  
24:        // define the exception classes  
25:        class xBoundary {};  
26:        class xSize {};  
27:        class xTooBig : public xSize {};  
28:        class xTooSmall : public xSize {};  
29:        class xZero : public xTooSmall {};  
30:        class xNegative : public xSize {};  
31:    private:  
32:        int *pType;  
33:        int itsSize;
```

```

34: };
35:
36:
37: Array::Array(int size):
38:     itsSize(size)
39: {
40:     if (size == 0)
41:         throw xZero();
42:     if (size > 30000)
43:         throw xTooBig();
44:     if (size < 1)
45:         throw xNegative();
46:     if (size < 10)
47:         throw xTooSmall();
48:
49:     pType = new int[size];
50:     for (int i = 0; i < size; i++)
51:         pType[i] = 0;
52: }
53:
54: int& Array::operator[](int offSet)
55: {
56:     int size = GetitsSize();
57:     if (offSet >= 0 && offSet < GetitsSize())
58:         return pType[offSet];
59:     throw xBoundary();
60:     return pType[0]; // appease MFC
61: }
62:
63: const int& Array::operator[](int offSet) const
64: {
65:     int mysize = GetitsSize();
66:
67:     if (offSet >= 0 && offSet < GetitsSize())
68:         return pType[offSet];
69:     throw xBoundary();
70:
71:     return pType[0]; // appease MFC
72: }
73:
74: int main()
75: {
76:     try
77:     {
78:         Array intArray(0);
79:         for (int j = 0; j < 100; j++)
80:         {
81:             intArray[j] = j;
82:             cout << "intArray[" << j << "] okay..." << endl;
83:         }
84:     }
85:     catch (Array::xBoundary)
86:     {
87:         cout << "Unable to process your input!" << endl;
88:     }
89:     catch (Array::xTooBig)
90:     {
91:         cout << "This array is too big..." << endl;
92:     }
93:
94:     catch (Array::xTooSmall)
95:     {
96:         cout << "This array is too small..." << endl;
97:     }
98:     catch (Array::xZero)
99:     {
100:         cout << "You asked for an array";
101:         cout << " of zero objects!" << endl;
102:     }
103:     catch (...)
104:     {
105:         cout << "Something went wrong!" << endl;
106:     }

```

```

107:     cout << "Done." << endl;
108:     return 0;
109: }

```

### ▼ 输出:

```

This array is too small...
Done.

```

### ▼ 分析:

重要修改在第 27~30 行, 这里建立了类层次结构。类 `xTooBig`、`xTooSmall` 和 `xNegative` 是从 `xSize` 派生而来的, 而 `xZero` 是从 `xTooSmall` 派生而来的。

创建了一个大小为 0 的 `Array`, 但这是什么呢? 看似捕获的异常是错误的! 然而, 仔细检查 `catch` 块将发现, 它在捕获 `xZero` 异常前捕获 `xTooSmall` 异常。由于引发的是 `xZero` 异常, 而 `xZero` 对象也是 `xTooSmall` 对象, 因此它被 `xTooSmall` 的处理程序捕获。被处理后, 异常将不会被传递给其他处理程序, 因此 `xZero` 的处理程序永远不会被调用。

这种问题的解决方案是, 仔细排列处理程序的顺序, 将具体的处理程序放在前面, 将不那么具体的处理程序放在后面。在这个例子中, 交换 `xZero` 和 `xTooSmall` 的处理程序的位置就可以解决问题。

## 28.5 异常中的数据及给异常对象命名

对于被引发的异常, 通常要知道除其类型之外的其他信息, 以便能够正确地应对错误。和其他类一样, 异常类也可以包含数据成员、在构造函数中初始化数据成员、在任何时候读取数据成员。程序清单 28.7 演示了这一点。

程序清单 28.7 读取异常对象中的数据

```

0: #include <iostream>
1: using namespace std;
2:
3: const int DefaultSize = 10;
4:
5: class Array
6: {
7:     public:
8:         // constructors
9:         Array(int itsSize = DefaultSize);
10:        Array(const Array &rhs);
11:        ~Array() { delete [] pType;}
12:
13:        // operators
14:        Array& operator=(const Array&);
15:        int& operator[](int offSet);
16:        const int& operator[](int offSet) const;
17:
18:        // accessors
19:        int GetitsSize() const { return itsSize; }
20:
21:        // friend function
22:        friend ostream& operator<< (ostream&, const Array&);
23:
24:        // define the exception classes
25:        class xBoundary {};
26:        class xSize
27:        {
28:            public:
29:            xSize(int size):itsSize(size) {}
30:            ~xSize(){}
31:            int GetSize() { return itsSize; }
32:            private:
33:            int itsSize;
34:        };
35:
36:        class xTooBig : public xSize

```

```
37:     {
38:         public:
39:             xTooBig(int size):xSize(size){}
40:     };
41:
42:     class xTooSmall : public xSize
43:     {
44:         public:
45:             xTooSmall(int size):xSize(size){}
46:     };
47:
48:     class xZero : public xTooSmall
49:     {
50:         public:
51:             xZero(int size):xTooSmall(size){}
52:     };
53:
54:     class xNegative : public xSize
55:     {
56:         public:
57:             xNegative(int size):xSize(size){}
58:     };
59:
60:     private:
61:         int *pType;
62:         int itsSize;
63:     };
64:
65:
66:     Array::Array(int size):
67:     itsSize(size)
68:     {
69:         if (size == 0)
70:             throw xZero(size);
71:         if (size > 30000)
72:             throw xTooBig(size);
73:         if (size < 1)
74:             throw xNegative(size);
75:         if (size < 10)
76:             throw xTooSmall(size);
77:
78:         pType = new int[size];
79:         for (int i = 0; i < size; i++)
80:             pType[i] = 0;
81:     }
82:
83:
84:     int& Array::operator[] (int offSet)
85:     {
86:         int size = GetitsSize();
87:         if (offSet >= 0 && offSet < size)
88:             return pType[offSet];
89:         throw xBoundary();
90:         return pType[0];
91:     }
92:
93:     const int& Array::operator[] (int offSet) const
94:     {
95:         int size = GetitsSize();
96:         if (offSet >= 0 && offSet < size)
97:             return pType[offSet];
98:         throw xBoundary();
99:         return pType[0];
100:     }
101:
102:     int main()
103:     {
104:         try
105:         {
106:             Array intArray(9);
107:             for (int j = 0; j < 100; j++)
108:             {
109:                 intArray[j] = j;
```



```

110:         cout << "intArray[" << j << "] okay..." << endl;
111:     }
112: }
113: catch (Array::xBoundary)
114: {
115:     cout << "Unable to process your input!" << endl;
116: }
117: catch (Array::xZero theException)
118: {
119:     cout << "You asked for an Array of zero objects!" << endl;
120:     cout << "Received " << theException.GetSize() << endl;
121: }
122: catch (Array::xTooBig theException)
123: {
124:     cout << "This Array is too big..." << endl;
125:     cout << "Received " << theException.GetSize() << endl;
126: }
127: catch (Array::xTooSmall theException)
128: {
129:     cout << "This Array is too small..." << endl;
130:     cout << "Received " << theException.GetSize() << endl;
131: }
132: catch (...)
133: {
134:     cout << "Something went wrong, but I've no idea what!\n";
135: }
136: cout << "Done." << endl;
137: return 0;
138: }

```

#### ▼ 输出:

```

This array is too small...
Received 9
Done.

```

#### ▼ 分析:

xSize 的声明被修改为包含一个成员变量 itsSize (第 33 行) 和一个成员函数 GetSize() (第 31 行)。另外, 添加了一个构造函数, 它接受一个 int 参数并初始化成员变量, 如第 29 行所示。

派生类声明了一个只初始化基类的构造函数。为节省篇幅, 没有声明其他函数。

第 113~135 行的 catch 语句修改为将它们捕获的异常命名为 theException, 并使用这个对象来访问存储在 itsSize 中的数据。

#### 注意

在发生异常 (出现某种错误) 后, 如果要创建异常对象, 应避免在创建它时引发同样的问题。因此, 如果要创建 OutOfMemory 异常, 则不应在其构造函数中分配内存。

让每条 catch 语句分别打印相应的消息很繁琐, 也容易出错。这种工作应由对象来完成, 它知道自己的类型和存储的值。程序清单 28.8 采用了一种面向对象程度更高的方法来解决该问题: 使用虚函数让每个异常都能“做正确的事情”。

#### 程序清单 28.8 按引用传递及在异常中使用虚函数

```

0: #include <iostream>
1: using namespace std;
2:
3: const int DefaultSize = 10;
4:
5: class Array
6: {
7:     public:
8:         // constructors
9:         Array(int itsSize = DefaultSize);
10:        Array(const Array &rhs);
11:        ~Array() { delete [] pType;}
12:

```

```
13: // operators
14: Array& operator=(const Array&);
15: int& operator[](int offSet);
16: const int& operator[](int offSet) const;
17:
18: // accessors
19: int GetitsSize() const { return itsSize; }
20:
21: // friend function
22: friend ostream& operator<<
23: (ostream&, const Array&);
24:
25: // define the exception classes
26: class xBoundary {};
27: class xSize
28: {
29: public:
30:     xSize(int size):itsSize(size) {}
31:     ~xSize(){}
32:     virtual int GetSize() { return itsSize; }
33:     virtual void PrintError()
34:     {
35:         cout << "Size error. Received: ";
36:         cout << itsSize << endl;
37:     }
38: protected:
39:     int itsSize;
40: };
41:
42: class xTooBig : public xSize
43: {
44: public:
45:     xTooBig(int size):xSize(size){}
46:     virtual void PrintError()
47:     {
48:         cout << "Too big! Received: ";
49:         cout << xSize::itsSize << endl;
50:     }
51: };
52:
53: class xTooSmall : public xSize
54: {
55: public:
56:     xTooSmall(int size):xSize(size){}
57:     virtual void PrintError()
58:     {
59:         cout << "Too small! Received: ";
60:         cout << xSize::itsSize << endl;
61:     }
62: };
63:
64: class xZero : public xTooSmall
65: {
66: public:
67:     xZero(int size):xTooSmall(size){}
68:     virtual void PrintError()
69:     {
70:         cout << "Zero!! Received: ";
71:         cout << xSize::itsSize << endl;
72:     }
73: };
74:
75: class xNegative : public xSize
76: {
77: public:
78:     xNegative(int size):xSize(size){}
79:     virtual void PrintError()
80:     {
81:         cout << "Negative! Received: ";
82:         cout << xSize::itsSize << endl;
83:     }
84: };
85:
```



```

86: private:
87:     int *pType;
88:     int itsSize;
89: };
90:
91: Array::Array(int size):
92:     itsSize(size)
93: {
94:     if (size == 0)
95:         throw xZero(size);
96:     if (size > 30000)
97:         throw xTooBig(size);
98:     if (size < 0)
99:         throw xNegative(size);
100:     if (size < 10)
101:         throw xTooSmall(size);
102:
103:     pType = new int[size];
104:     for (int i = 0; i < size; i++)
105:         pType[i] = 0;
106: }
107:
108: int& Array::operator[] (int offSet)
109: {
110:     int size = GetitsSize();
111:     if (offSet >= 0 && offSet < GetitsSize())
112:         return pType[offSet];
113:     throw xBoundary();
114:     return pType[0];
115: }
116:
117: const int& Array::operator[] (int offSet) const
118: {
119:     int size = GetitsSize();
120:     if (offSet >= 0 && offSet < GetitsSize())
121:         return pType[offSet];
122:     throw xBoundary();
123:     return pType[0];
124: }
125:
126: int main()
127: {
128:     try
129:     {
130:         Array intArray(9);
131:         for (int j = 0; j < 100; j++)
132:         {
133:             intArray[j] = j;
134:             cout << "intArray[" << j << "] okay..." << endl;
135:         }
136:     }
137:     catch (Array::xBoundary)
138:     {
139:         cout << "Unable to process your input!" << endl;
140:     }
141:     catch (Array::xSize& theException)
142:     {
143:         theException.PrintError();
144:     }
145:     catch (...)
146:     {
147:         cout << "Something went wrong!" << endl;
148:     }
149:     cout << "Done." << endl;
150:     return 0;
151: }

```

#### ▼ 输出:

```

Too small! Received: 9
Done.

```

## ▼ 分析:

在程序清单 28.8 中, 第 33~37 行为 xSize 类中声明了虚方法 PrintError(), 它打印一条错误消息和对象的大小 (itsSize)。在每个派生类中都覆盖了该方法。

第 141 行将异常处理程序声明为接受一个异常对象引用。通过对象引用来调用 PrintError() 时, 多态使得正确的 PrintError() 版本被调用。代码更清晰, 更易于理解和维护。

## 28.6 异常和模板

要在模板中使用异常, 可以为模板的每个实例创建一个异常类, 也可以使用在模板外声明的异常类。程序清单 28.9 演示了这 2 种方式。

程序清单 28.9 在模板中使用异常

```
0: #include <iostream>
1: using namespace std;
2:
3: const int DefaultSize = 10;
4: class xBoundary {};
5:
6: template <class T>
7: class Array
8: {
9:     public:
10:         // constructors
11:         Array(int itsSize = DefaultSize);
12:         Array(const Array &rhs);
13:         ~Array() { delete [] pType;}
14:
15:         // operators
16:         Array& operator=(const Array<T>&);
17:         T& operator[](int offSet);
18:         const T& operator[](int offSet) const;
19:
20:         // accessors
21:         int GetitsSize() const { return itsSize; }
22:
23:         // friend function
24:         friend ostream& operator<< (ostream&, const Array<T>&);
25:
26:         // define the exception classes
27:
28:         class xSize {};
29:
30:     private:
31:         int *pType;
32:         int itsSize;
33: };
34:
35: template <class T>
36: Array<T>::Array(int size):
37:     itsSize(size)
38: {
39:     if (size < 10 || size > 30000)
40:         throw xSize();
41:     pType = new T[size];
42:     for (int i = 0; i < size; i++)
43:         pType[i] = 0;
44: }
45:
46: template <class T>
47: Array<T>& Array<T>::operator=(const Array<T> &rhs)
48: {
49:     if (this == &rhs)
50:         return *this;
51:     delete [] pType;
52:     itsSize = rhs.GetitsSize();
```

```

53:     pType = new T[itsSize];
54:     for (int i = 0; i < itsSize; i++)
55:         pType[i] = rhs[i];
56: }
57: template <class T>
58: Array<T>::Array(const Array<T> &rhs)
59: {
60:     itsSize = rhs.GetitsSize();
61:     pType = new T[itsSize];
62:     for (int i = 0; i < itsSize; i++)
63:         pType[i] = rhs[i];
64: }
65:
66: template <class T>
67: T& Array<T>::operator[](int offSet)
68: {
69:     int size = GetitsSize();
70:     if (offSet >= 0 && offSet < GetitsSize())
71:         return pType[offSet];
72:     throw xBoundary();
73:     return pType[0];
74: }
75:
76: template <class T>
77: const T& Array<T>::operator[](int offSet) const
78: {
79:     int mysize = GetitsSize();
80:     if (offSet >= 0 && offSet < GetitsSize())
81:         return pType[offSet];
82:     throw xBoundary();
83: }
84:
85: template <class T>
86: ostream& operator<< (ostream& output, const Array<T>& theArray)
87: {
88:     for (int i = 0; i < theArray.GetitsSize(); i++)
89:         output << "[" << i << "]" << theArray[i] << endl;
90:     return output;
91: }
92:
93:
94: int main()
95: {
96:     try
97:     {
98:         Array<int> intArray(9);
99:         for (int j = 0; j < 100; j++)
100:         {
101:             intArray[j] = j;
102:             cout << "intArray[" << j << "] okay..." << endl;
103:         }
104:     }
105:     catch (xBoundary)
106:     {
107:         cout << "Unable to process your input!" << endl;
108:     }
109:     catch (Array<int>::xSize)
110:     {
111:         cout << "Bad Size!" << endl;
112:     }
113:     cout << "Done." << endl;
114:     return 0;
115: }
116: }

```

#### ▼ 输出:

```

Bad Size!
Done.

```

#### ▼ 分析:

第 1 个异常类 (xBoundary) 是在模板定义外 (第 4 行) 声明的。第 2 个异常类 xSize 是在模板定

义内（第28行）声明的。

异常类 `xBoundary` 没有和模板类捆绑在一起，但可以像使用其他类那样使用它。`xSize` 和模板捆绑在一起，必须通过 `Array` 实例来调用。正如读者看到的，两条 `catch` 语句的语法不同：第105行为 `catch (xBoundary)`，第109行为 `catch (Array<int>::xSize)`。后者与 `int` 型 `Array` 实例捆绑在一起。

## 28.7 没有错误的异常

C++程序员工作之余聚集在网吧聊天时，常常会聊到这样一个话题：异常是否应用于常见状态。一些人坚持认为，就其本质而言，异常只应用于那些可预见的、程序员必须预先考虑的异常情形，而不是代码中正常处理的部分。

另一些人指出，异常提供了强大而清晰的返回方式，可跨越多层函数调用而没有内存泄漏的危险。一个常见的例子是：用户在图形用户界面（GUI）环境中请求执行操作。捕获这种请求的代码必须调用对话框管理器的一个成员函数，该成员函数调用处理请求的代码，处理请求的代码调用决定使用哪个对话框的代码，这些代码又调用创建对话框的代码，而后者调用处理用户输入的代码。如果用户单击“取消”按钮，程序必须返回处理最初请求的第一个调用方法。

解决这种问题的一种方法是，将初始调用放在一个 `try` 块中，并将 `CancelDialog` 作为异常，由“取消”按钮的处理程序引发。这是安全有效的，但单击“取消”按钮是正常情形，而不是异常情形。

这常常会变得有点像宗教争论，但解决争端的合理方式是提出如下问题：以这种方式使用异常会使代码更容易还是更难理解？出现错误和内存泄露的风险更小还是更大？维护这种代码更容易还是更难？像其他决策一样，这些决策也要求折衷，没有唯一的显而易见的正确答案。

### 代码蜕变简介

代码蜕变（Code Rot）是一种众所周知的现象，指的是软件由于缺乏维护而恶化。编写得很好、经过充分调试的程序在交付几个星期后，在用户的系统上变坏了。几个月后，用户将发现程序逻辑被破坏，很多对象开始剥落。

除将源代码放在密闭的容器中外，唯一可采取的防护措施是，编写程序时确保以后修复时能够快速、轻松地找到问题所在。

代码蜕变是程序员的一个玩笑，用于说明一个重要的经验教训。程序是极其复杂的，各种错误可能潜伏很长一段时间才暴露出来。保护措施是编写易于维护的代码。

这意味着编写的代码必须易于理解，对微妙的地方进行注释。交付6个月后再阅读自己编写的代码时，您将感到完全陌生，奇怪竟然有人编写了如此晦涩难懂的逻辑。

## 28.8 bug 和调试

几乎所有的现代开发环境都包括一个或多个功能强大的调试器。使用调试器的基本思想是：首先运行调试器，它加载源代码，然后在调试器中运行程序。这让程序员能够看到程序中每条指令的执行情况以及检查变量在程序执行期间的变化情况。

所有的编译器都允许使用或不使用符号（symbol）进行编译。使用符号的编译命令编译器在源代码和生成的程序之间建立必要的映射关系；调试器根据映射关系指出程序的下一个操作的对应的源代码行。

全屏幕符号调试器使这项繁琐的工作变得轻松愉快。加载调试器时，它将读取所有的源代码，并将其显示在窗口中。您可以将函数调用作为一步执行，也可以命令调试器进入函数，逐行地执行其中的代码。

在大多数调试器中，可以在源代码和输出之间切换，查看每条语句的执行结果。更为强大的功能是，可以查看每个变量的当前状态，查看复杂的数据结构，查看类中成员数据的值，查看各种指针指

向的内存和其他内存单元中的实际值。可以在调试器中使用多种控制方式，包括设置断点、设置监视点、查看内存和汇编代码。

### 28.8.1 断点

断点命令调试器在到达某行代码后暂停执行程序。这让程序一直运行到被怀疑的代码行。断点可帮助您分析在关键代码行执行前后，变量的当前状态。

### 28.8.2 监视点

可让调试器显示某个变量的值或某个变量被读写时暂停程序执行。监视点让您能够设置这些条件，甚至在程序运行时修改变量的值。

### 28.8.3 查看内存

有时查看内存中存储的实际值很重要。现代调试器可以用实际变量的格式显示值，即将字符串显示为字符、长整数显示为数字而不是 4 个字节的内容等。高级 C++ 调试器甚至能够显示整个对象，提供包括 this 指针在内的所有成员变量的当前值。

### 28.8.4 查看汇编代码

虽然要发现 bug，只需查看源代码即可，但当其他手段都不管用时，可以命令调试器显示根据每行源代码生成的汇编代码。可以查看内存寄存器和标记，并在必要时深入研究程序的内部工作原理。

读者应学会使用调试器。它是战胜 bug 的最强有力武器。运行阶段错误是最难发现和消除的，强大的调试器让您几乎可能找到所有的错误，虽然查找起来可能并不那么轻松。

## 28.9 总结

本章介绍了有关创建和使用异常的基本知识。异常是可以在程序的某些地方创建并引发的对象，在这些地方，执行的代码不能处理错误或发生了其他异常情况。程序的其他部分（调用堆栈的上方）实现了 catch 块，它能够捕获异常并采取适当的措施。

异常是用户创建的常规对象，可以按值或引用进行传递。它们可以包含数据和方法，catch 块可以根据这些数据决定如何处理异常。

可以创建多个 catch 块，但异常与某个 catch 块匹配后，就被视为已被处理，不会被传递给后面的 catch 块。应按正确的顺序排列 catch 块，将具体的 catch 块放在前面，让通用的 catch 块处理那些其他 catch 块不能处理的异常，这非常重要。

本章还介绍了有关符号调试器的基本知识，包括使用监视点和断点等。这些工具让您能够将注意力集中到程序中导致错误的部分，查看程序执行期间变量的变化情况。

## 28.10 问与答

问：为什么要引发异常？为什么不就地处理错误？

答：通常，在程序的不同地方可能导致相同的错误。异常让您能够集中处理这些错误。另外，导致错误的地方可能不是决定如何处理错误的最佳位置。

问：为什么要生成异常对象？为什么不只传递错误码？

答：异常对象比错误码功能更强大、更灵活。它们能够传递更多的信息，同时可以使用构造函数/析构函数机制来分配和释放正确处理异常所需的资源。

问：为什么不将异常用于非错误状态？能够快速返回以前的代码区域，即使在非异常状态下也是如此，不是很方便吗？

答：是的。有些 C++ 程序员仅为此目而使用异常。这样做的危险是，当堆栈被解退时会无意中某些对象留在自由存储区中，导致内存泄漏。使用细致的编程技术和优秀的编译器，通常可以避免这种情况。然而，这是个人审美观问题，有些程序员认为，异常从其本质上说不能用于正常状态。

问：必须在引发异常的 try 块后面捕获它的吗？

答：不，可以在调用栈的任何地方捕获异常。当堆栈被解退时，异常传递将沿堆栈向上传递，直到被处理为止。

问：在可以 cout 和诸如此类的其他语句的情况下，为什么要使用调试器？

答：调试器提供了一种功能强大得多的机制，让您能够以步进方式执行程序并监视变量的变化情况，而无需使用成千上万条的调试语句，使代码混乱不堪。另外，每次添加和删除代码行时都会带来一定的风险。如果您调试的目的只是为了修复问题，但在删除添加的 cout 语句时不小心删除了真正的代码行，将带来新的问题。

## 28.11 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 28.11.1 测验

1. 什么是异常？
2. 什么是 try 块？
3. 什么是 catch 语句？
4. 异常对象可包含什么信息？
5. 异常对象在什么时候被创建？
6. 应按值还是引用来传递异常对象？
7. 如果 catch 语句捕获基类异常对象，它能捕获到派生类异常对象吗？
8. 如果使用了两条 catch 语句，一条捕获基类异常对象，另一条捕获派生类异常对象，应将哪条语句放在前面？
9. catch(...) 的含义是什么？
10. 什么是断点？

### 28.11.2 练习

1. 编写一个 try 块、一条 catch 语句和一个简单异常。
2. 修改练习 1 的答案，在异常类中添加数据和一个存取器函数，并在 catch 块中使用它。



3. 将练习 2 中的类修改为异常类层次结构。修改 catch 块，使其捕获派生类对象和基类对象。
4. 修改练习 3 中的程序，使其包含三级函数调用。
5. 查错：下面的代码有什么错误？

```
#include "stringc.h"          // our string class

class xOutOfMemory
{
public:
    xOutOfMemory( const String& where ) : location( where ){}
    ~xOutOfMemory(){}
    virtual String where(){ return location };
private:
    String location;
}

int main()
{
    try
    {
        char *var = new char;
        if ( var == 0 )
            throw xOutOfMemory();
    }
    catch( xOutOfMemory& theException )
    {
        cout << "Out of memory at " << theException.location() << endl;
    }
    return 0;
}
```

该程序清单演示了如何处理内存耗尽错误。

## 第 29 章

# 杂项内容

首先祝贺您！您就要完成全面介绍 C++ 的课程了。现在，您对 C++ 有深入了解，但现代编程中，总是有更多的知识需要学习。本章补充一些遗漏的细节，然后为继续学习提供一些建议。

您在源代码文件中编写的大部分代码都是 C++。它由编译器编译并生成可执行程序。然而，在编译器运行之前将运行预处理器，这提供了条件编译的机会。

在本章中，您将学习：

- 什么是条件编译？如何管理条件编译
- 如何在查找错误时使用预处理器
- 如何操纵位以及将它们用作标记
- 高效使用 C++ 的后续步骤

### 29.1 预处理器和编译器

每次运行编译器时，预处理器都将首先运行。预处理器查找预处理器指令，每条预处理器指令都以 # 打头。这些指令的作用是修改源代码的文本。结果为一个新的源代码文件：一个通常看不到的临时文件，但您可以命令编译器保存它，以便在需要时查看。

编译器不是读取原始的源代码文件，而是读取预处理器的输出并对其进行编译。读者已经通过使用编译指令 `#include` 看到了这种效果。`#include` 命令预处理器查找其名称位于 `#include` 后面的文件，并将其写入到中间文件这个位置。这就像您将整个文件的内容输入到了源代码中一样，编译器看到源代码时，包含的文件内容已经在源代码中了。

#### 提示

几乎每个编译器都有一个可在集成开发环境（IDE）或命令行中设置的开关，用于指示编译器保存中间文件。如果要查看中间文件，可参阅编译器手册来了解需要为编译器设置哪个开关。

### 29.2 预编译器指令 `#define`

可使用命令 `#define` 来定义字符串替换，下面的代码指示预处理器将所有的“BIG”替换为 512：

```
#define BIG 512
```

这不是 C++ 意义上的字符串。源代码中所有的“BIG”都将被替换为“512”。对于如下源代码：

```
#define BIG 512
int myArray[BIG];
```

经预处理器处理后将变成：

```
int myArray[512];
```

注意到 `#define` 语句不见了。在中间文件中，预编译语句都将被删除，它们根本不会出现在最终的源代码中。

### 29.2.1 使用#define 定义常量

#define 的用途之一是定义常量。然而，由于#define 只是进行字符串替换而不做类型检查，因此这几乎不是什么好主意。正如在介绍常量时指出的，相对于使用#define，使用关键字 const 有很多优点。

### 29.2.2 将#define 用于检测

#define 的另一种用途是，指出某个字符串定义过。可以编写这样的代码：

```
#define DEBUG
```

然后在程序中检测 DEBUG 是否被定义，并采取相应的措施。要检查字符串是否被定义过，可使用预编译器命令 #if 和命令 defined：

```
#if defined DEBUG
cout << "Debug defined";
#endif
```

如果检测的字符串（这里为 DEBUG）已定义，则 defined 表达式的结果为 true。别忘了，这发生在预处理器中，而不是编译器中或执行程序时。

预编译器遇到 #if defined 时，将检查一个已创建的表，看 #if defined 后面的值是否定义了。如果定义了，则 defined 表达式的结果为 true，#if defined DEBUG 和 #endif 之间的所有内容都将被写入中间文件进行编译；如果为 false，#if defined DEBUG 和 #endif 之间的所有内容都不会写入中间文件，就好像它们本来就不在源代码中一样。

还有一个简化的编译指令可用于检测某个值是否被定义，这就是 #ifdef：

```
#ifdef DEBUG
cout << "Debug defined";
#endif
```

还可以检测某个值是否未定义，为此可将运算符 ! 用于编译指令 defined：

```
#if !defined DEBUG
cout << "Debug is not defined";
#endif
```

也可使用其简化版本 #ifndef：

```
#ifndef DEBUG
cout << "Debug is not defined.";
#endif
```

注意，#ifndef 是 #ifdef 的逻辑逆，如果在此之前没有定义被检测的字符串，则 #ifndef 的结果为 true。读者应该注意到了，这些检测都需要使用 #endif 来指定受检测影响的代码到什么地方结束。

### 29.2.3 预编译器命令 #else

读者可能想到了，可在 #ifdef（或 #ifndef）和 #endif 之间使用编译指令 #else，程序清单 29.1 演示了如何使用这些编译指令。

程序清单 29.1 使用#define

```
0: #define DemoVersion
1: #define SW_VERSION 5
2: #include <iostream>
3:
4: using std::endl;
5: using std::cout;
6:
7: int main()
8: {
9:     cout << "Checking on the definitions of DemoVersion,";
10:    cout << "SW_VERSION, and WINDOWS_VERSION..." << endl;
```

```

11:
12:     #ifdef DemoVersion
13:         cout << "DemoVersion defined." << endl;
14:     #else
15:         cout << "DemoVersion not defined." << endl;
16:     #endif
17:
18:     #ifndef SW_VERSION
19:         cout << "SW_VERSION not defined!" << endl;
20:     #else
21:         cout << "SW_VERSION defined as: "
22:             << SW_VERSION << endl;
23:     #endif
24:
25:     #ifdef WINDOWS_VERSION
26:         cout << "WINDOWS_VERSION defined!" << endl;
27:     #else
28:         cout << "WINDOWS_VERSION was not defined." << endl;
29:     #endif
30:
31:     cout << "Done." << endl;
32:     return 0;
33: }

```

#### ▼ 输出:

```

Checking on the definitions of DemoVersion, NT_VERSION, and WINDOWS_VERSION...
DemoVersion defined.
NT_VERSION defined as: 5
WINDOWS_VERSION was not defined.
Done.

```

#### ▼ 分析:

第 0 行和第 1 行定义了 `DemoVersion` 和 `SW_VERSION`，其中后者被定义为 5。第 12 行检测 `DemoVersion` 是否已定义。由于 `DemoVersion` 已定义（虽然没有值），因此检测结果为 `true`，并执行第 13 行的代码。

第 18 行检测 `SW_VERSION` 是否未定义。由于 `SW_VERSION` 已定义，因此检测结果为 `false`，并跳到第 21 行执行。在这里，`SW_VERSION` 已被替换为 5，编译器看到的代码如下：

```
cout << "SW_VERSION defined as: " << 5 << endl;
```

注意，第一个 `SW_VERSION` 没有被替换，因为它位于用引号括起的字符串中，然而，第二个 `SW_VERSION` 被替换，因此编译器看到的是 5，就像在这里输入了 5 一样。

最后在第 25 行检测 `WINDOWS_VERSION` 是否已定义。由于没有定义 `WINDOWS_VERSION`，因此检测结果为 `false`，执行第 28 行打印一条消息。

## 29.3 包含和防范多重包含

创建项目时将使用很多不同的文件。可能需要组织目录，使每个类都有包含类声明的头文件（如 `.hpp`）和包含类方法源代码的实现文件（如 `.cpp`）。

`main()` 函数保存在一个独立的 `.cpp` 文件中，所有 `.cpp` 文件都被编译为 `.obj` 文件，然后由链接器将它们链接成一个程序。

由于程序使用了很多类的方法，因此很多头文件都将被包含到每个文件中。另外，头文件经常需要包含其他头文件。例如，派生类的头文件必须包含基类的头文件。

假设 `Animal` 类是在文件 `ANIMAL.hpp` 中声明的。在从 `Animal` 类派生而来的 `Dog` 类的头文件 `DOG.hpp` 中，必须包含头文件 `ANIMAL.hpp`，否则将不能从 `Animal` 派生出 `Dog`。鉴于同样的原因，`Cat` 类的头文件也必须包含 `ANIMAL.hpp`。

创建使用了 `Cat` 和 `Dog` 的程序时，可能将 `ANIMAL.hpp` 包含两次。这将导致编译错误，因为不能

将同一个类 (Animal) 声明两次, 虽然这两次声明是相同的。

为解决这种问题, 可使用多重包含防范 (inclusion guard)。在 ANIMAL 的头文件开头, 添加如下代码行:

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
...           // the whole file goes here
#endif
```

上述代码的含义是, 如果没有定义 ANIMAL\_HPP, 则现在定义它。在 #define 语句和 #endif 语句之间是整个文件的内容。

程序首次包含该文件时, 读取第 1 行, 检测结果为 true: 还没有定义 ANIMAL\_HPP, 因此定义它并包含整个文件。

程序第二次包含文件 ANIMAL\_HPP 时, 它读取第 1 行, 检测结果为 false, 因为已经包含了 ANIMAL\_HPP。因此预处理器不处理下一个 #else (在这个例子中没有) 或 #endif (在文件末尾) 之前的所有代码。这样就跳过了整个文件的内容, 类不会被声明两次。

符号的实际名称 (ANIMAL\_HPP) 并不重要, 虽然习惯上使用全部大写的文件名, 并将句点 (.) 改为下划线。然而, 这纯粹是一种约定, 由于两个文件不能同名, 因此这种约定是可行的。

#### 注意

使用多重包含防范不会有任何害处。使用它常常可以节省大量的调试时间。

## 29.4 字符串操纵

预处理器提供了两个特殊的运算符, 用于在宏中操纵字符串。字符串化运算符 (#) 将其后面的内容转换为用引号括起的字符串; 拼接运算符将两个字符串合并成一个。

### 29.4.1 字符串化

字符串化运算符将其后面到下一个空格为止的所有字符用引号括起。如果编写了如下宏:

```
#define WRITESTRING(x) cout << #x
```

然后这样调用它:

```
WRITESTRING(This is a string);
```

预编译器将把它转换为:

```
cout << "This is a string";
```

注意, 按 cout 的要求将字符串 This is a string 用引号括起。

### 29.4.2 拼接

拼接运算符让您能够将多个单词合并成一个新词。该新词实际上是一个符号, 可用作类名、变量名、数组下标或出现任何可使用字符串的地方。

假设有 5 个函数, 分别名为 fOnePrint、fTwoPrint、fThreePrint、fFourPrint 和 fFivePrint。可以这样声明一个宏:

```
#define fPRINT(x) f ## x ## Print
```

然后, 使用 fPRINT(Two)来生成 fTwoPrint, 使用 fPRINT(Three)来生成 fThreePrint。

## 29.5 预定义的宏

很多编译器预定义了大量有用的宏, 其中包括 \_\_DATE\_\_、\_\_TIME\_\_、\_\_LINE\_\_ 和 \_\_FILE\_\_ 每个宏

名都以两个下划线字符开头和结尾，以降低这些宏名与程序员在程序中使用的名称发生冲突的可能性。

预编译器看到这些宏时，将执行合适的替换。对于 `__DATE__`，替换为当前日期；对于 `__TIME__`，替换为当前时间；对于 `__LINE__` 和 `__FILE__`，分别替换为源代码行数和文件名。需要注意的是，这些替换是在预编译源代码时进行的，而不是在程序运行时进行的。如果要求程序打印 `__DATE__`，打印的将不是当前日期，而是程序被编译时的日期。这些预定义的宏在记录并分析问题代码方面很有用。

## 29.6 assert( )宏

很多编译器都提供了一个 `assert( )` 宏。参数的值为 `true` 时，`assert( )` 返回 `true`；否则执行某种操作。很多编译器在 `assert( )` 的参数值为 `false` 时中止程序，其他编译器则引发异常。

`assert( )` 宏用于在发布程序前对其进行调试。事实上，如果 `DEBUG` 没有定义，预处理器将简化 `assert( )`，使得其中的任何代码都不会出现在为编译器生成的源代码中。这在开发过程中很有帮助，发布最终产品时，`assert( )` 不会影响性能，也不增加程序可执行版本的大小。

也可以使用编译器提供的 `assert( )`，而编写自己的 `assert( )` 宏。程序清单 29.2 提供了一个简单的自定义 `assert( )` 宏并演示了其用法。

程序清单 29.2 一个简单的 `assert( )` 宏

---

```

0: // Listing 29.2 ASSERTS
1: #define DEBUG
2: #include <iostream>
3: using namespace std;
4:
5: #ifndef DEBUG
6:     #define ASSERT(x)
7: #else
8:     #define ASSERT(x) \
9:         if (! (x)) \
10:        { \
11:            cout << "ERROR!! Assert " << #x << " failed << endl; \
12:            cout << " on line " << __LINE__ << endl; \
13:            cout << " in file " << __FILE__ << endl; \
14:        }
15: #endif
16:
17: int main()
18: {
19:     int x = 5;
20:     cout << "First assert: " << endl;
21:     ASSERT(x==5);
22:     cout << "\nSecond assert: " << endl;
23:     ASSERT(x != 5);
24:     cout << "\nDone. << endl";
25:     return 0;
26: }
```

---

### ▼ 输出:

```

First assert:

Second assert:
ERROR!! Assert x !=5 failed
on line 24
in file List2104.cpp

Done.
```

### ▼ 分析:

第 1 行定义了 `DEBUG`。通常，这是编译阶段通过命令行（或 IDE）完成的，这样可以根据需要打开或关闭它。第 8~14 行定义了 `ASSERT( )` 宏。通常，这是在头文件中完成的，在所有实现文件中都



应包含该头文件 (ASSERT.hpp)。

第 5 行检测 DEBUG 是否已定义。如果 DEBUG 没有定义, ASSERT() 被定义为不创建任何代码。如果定义了 DEBUG, 则应用第 8~14 行定义的功能。

对预编译器来说, ASSERT() 本身是一条很长的语句, 它跨越 7 行。第 9 行检测通过参数传入的值, 如果为 false, 则执行第 11~13 行的语句, 打印一条错误消息; 如果传入的值为 true, 则不执行任何操作。

### 29.6.1 使用 assert() 进行调试

编写程序时, 通常知道某些事情是真的: 函数有特定的值、指针是有效的等。bug 的本质是, 在某些情况下本为真的事实确是假的。例如, 您知道某个指针是有效的, 但程序却崩溃了。assert() 可帮助您发现这类 bug, 但仅当您养成在代码中大量使用 assert() 的习惯时才能如此。每当您给指针赋值、将其作为函数参数或返回值时, 务必确定该指针是有效的。每当代码依赖于某个变量包含的特定值时, 都应使用 assert() 来确认这一点。

频繁使用 assert() 没有坏处。解除对 DEBUG 的定义后, assert() 将从代码中删除。它还提供了优秀的内部文档, 提醒阅读者在程序执行的某个时刻, 您认为是真的事实。

### 29.6.2 assert() 与异常之比较

前一章介绍了如何使用异常处理错误状态。需要指出的是, assert() 并非为处理运行阶段的错误状态 (如输入无效、内存不足、不能打开文件等) 而设计的, 而是为捕获编程错误而设计的。也就是说, 如果 assert() “开火了”, 说明代码中有 bug。

这非常重要, 因为将代码交付给用户时, assert() 实例将被删除。不能依靠 assert() 来处理运行阶段的问题, 因为此时 assert() 已不存在了。

一种常见的错误是, 使用 assert() 来测试内存分配的返回值:

```
Animal *pCat = new Cat;  
Assert(pCat);    // bad use of assert  
pCat->SomeFunction();
```

这是一种经典的编程错误。每当程序员运行程序时, 有足够的内存可用, assert() 从不 “开火”。毕竟, 为提高编译器、调试器等速度, 程序员在安装了大量 RAM 的机器上运行程序。程序员然后发布可执行文件, 而可怜的用户没有那么多内存, 当执行到这部分时, 调用 new 失败并返回 NULL。然而, 代码中不再有 assert(), 没有任何东西指出指针为 NULL。执行到语句 pCat->SomeFounction() 后, 程序将崩溃。

虽然内存分配返回 NULL 是一种异常情形, 但并不是编程错误。程序只需引发异常就能够恢复。请切记: DEBUG 未被定义时, assert() 语句将消失。异常在第 28 章详细介绍过。

### 29.6.3 副作用

仅当 assert() 实例被删除后才出现的 bug 很常见。这几乎总是由于程序无意中依赖了 assert() 和其他调试代码的副作用引起的。例如, 如果本想检测 x 是否等于 5 时编写了如下代码:

```
ASSERT (x = 5)
```

将引入一个非常难以发现的 bug。

假设在这个 assert() 前, 您调用了将 x 设置为 0 的函数。您以为该 assert() 检测 x 是否等于 5, 而实际上却将 x 设置为 5。测试结果为 true, 因为 x=5 不仅将 x 设置为 5, 还返回 5, 由于 5 不等于零, 因此为 true。

执行完该 `assert()` 语句后, `x` 确实等于 5 (刚设置的!)。程序运行得很好, 您打算交付, 于是关闭调试。现在 `assert()` 消失了, 不再将 `x` 设置为 5。由于在此之前 `x` 刚刚被设置为 0, 因此它仍然为 0, 程序中断。

倍感挫折后, 您又打开调试, 但就像变魔术一样, `bug` 不见了。这看上去很有趣, 但还是经不住考验, 因此要当心调试代码的副作用。如果发现仅在调试关闭后才出现的 `bug`, 应查看调试代码, 注意讨厌的副作用。

## 29.6.4 类的不变量

大多数类都有一些条件, 每当执行完成成员函数后, 这些条件都将满足。这些类不变量是类的要素。例如, `Circle` 对象的半径不能为 0, `Animal` 的年龄总是在 0 和 100 之间。

声明一个这样的 `Invariants()` 方法可能会很有帮助: 它仅在所有这些条件都为 `true` 时才返回 `true`。这样, 可以在调用每个类方法之前和之后使用 `ASSERT(Invariants())`。例外情况是, 在构造函数执行前和析构函数结束后, `Invariants()` 将不会返回 `true`。程序清单 29.3 演示了如何在一个小型类中使用 `Invariants()`。

程序清单 29.3 使用 `Invariants()`

```
0: #define DEBUG
1: #define SHOW_INVARIANTS
2: #include <iostream>
3: #include <string.h>
4: using namespace std;
5:
6: #ifndef DEBUG
7:     #define ASSERT(x)
8: #else
9:     #define ASSERT(x) \
10:         if (! (x)) \
11:         { \
12:             cout << "ERROR!! Assert " << #x << " failed" << endl; \
13:             cout << " on line " << __LINE__ << endl; \
14:             cout << " in file " << __FILE__ << endl; \
15:         }
16: #endif
17:
18:
19: const int FALSE = 0;
20: const int TRUE = 1;
21: typedef int BOOL;
22:
23:
24: class String
25: {
26:     public:
27:         // constructors
28:         String();
29:         String(const char *const);
30:         String(const String &);
31:         ~String();
32:
33:         char & operator[](int offset);
34:         char operator[](int offset) const;
35:
36:         String & operator= (const String &);
37:         int GetLen()const { return itsLen; }
38:         const char * GetString() const { return itsString; }
39:         BOOL Invariants() const;
40:
41:     private:
42:         String (int);           // private constructor
43:         char * itsString;
44:         // unsigned short itsLen;
45:         int itsLen;
46: };
```

```
47:
48: // default constructor creates string of 0 bytes
49: String::String()
50: {
51:     itsString = new char[1];
52:     itsString[0] = '\0';
53:     itsLen=0;
54:     ASSERT(Invariants());
55: }
56:
57: // private (helper) constructor, used only by
58: // class methods for creating a new string of
59: // required size. Null filled.
60: String::String(int len)
61: {
62:     itsString = new char[len+1];
63:     for (int i = 0; i <= len; i++)
64:         itsString[i] = '\0';
65:     itsLen=len;
66:     ASSERT(Invariants());
67: }
68:
69: // Converts a character array to a String
70: String::String(const char * const cString)
71: {
72:     itsLen = strlen(cString);
73:     itsString = new char[itsLen+1];
74:     for (int i = 0; i < itsLen; i++)
75:         itsString[i] = cString[i];
76:     itsString[itsLen]='\0';
77:     ASSERT(Invariants());
78: }
79:
80: // copy constructor
81: String::String (const String & rhs)
82: {
83:     itsLen=rhs.GetLen();
84:     itsString = new char[itsLen+1];
85:     for (int i = 0; i < itsLen;i++)
86:         itsString[i] = rhs[i];
87:     itsString[itsLen] = '\0';
88:     ASSERT(Invariants());
89: }
90:
91: // destructor, frees allocated memory
92: String::~String ()
93: {
94:     ASSERT(Invariants());
95:     delete [] itsString;
96:     itsLen = 0;
97: }
98:
99: // operator equals, frees existing memory
100: // then copies string and size
101: String& String::operator=(const String & rhs)
102: {
103:     ASSERT(Invariants());
104:     if (this == &rhs)
105:         return *this;
106:     delete [] itsString;
107:     itsLen=rhs.GetLen();
108:     itsString = new char[itsLen+1];
109:     for (int i = 0; i < itsLen;i++)
110:         itsString[i] = rhs[i];
111:     itsString[itsLen] = '\0';
112:     ASSERT(Invariants());
113:     return *this;
114: }
115:
116: //non constant offset operator
117: char & String::operator[](int offset)
118: {
119:     ASSERT(Invariants());
```

```
120:     if (offset > itsLen)
121:     {
122:         ASSERT(Invariants());
123:         return itsString[itsLen-1];
124:     }
125:     else
126:     {
127:         ASSERT(Invariants());
128:         return itsString[offset];
129:     }
130: }
131:
132: // constant offset operator
133: char String::operator[](int offset) const
134: {
135:     ASSERT(Invariants());
136:     char retVal;
137:     if (offset > itsLen)
138:         retVal = itsString[itsLen-1];
139:     else
140:         retVal = itsString[offset];
141:     ASSERT(Invariants());
142:     return retVal;
143: }
144:
145: BOOL String::Invariants() const
146: {
147:     #ifdef SHOW_INVARIANTS
148:         cout << "String Tested OK ";
149:     #endif
150:     return ( (itsLen && itsString) || (!itsLen && !itsString) );
151: }
152:
153: class Animal
154: {
155:     public:
156:         Animal():itsAge(1),itsName("John Q. Animal")
157:             {ASSERT(Invariants());}
158:         Animal(int, const String&);
159:         ~Animal(){}
160:         int GetAge() { ASSERT(Invariants()); return itsAge;}
161:         void SetAge(int Age)
162:         {
163:             ASSERT(Invariants());
164:             itsAge = Age;
165:             ASSERT(Invariants());
166:         }
167:         String& GetName()
168:         {
169:             ASSERT(Invariants());
170:             return itsName;
171:         }
172:         void SetName(const String& name)
173:         {
174:             ASSERT(Invariants());
175:             itsName = name;
176:             ASSERT(Invariants());
177:         }
178:         BOOL Invariants();
179:     private:
180:         int itsAge;
181:         String itsName;
182: };
183:
184: Animal::Animal(int age, const String& name):
185:     itsAge(age),
186:     itsName(name)
187: {
188:     ASSERT(Invariants());
189: }
190:
191: BOOL Animal::Invariants()
192: {
```

```

193:     #ifdef SHOW_INVARIANTS
194:         cout << "Animal Tested OK";
195:     #endif
196:     return (itsAge > 0 && itsName.GetLen());
197: }
198:
199: int main()
200: {
201:     Animal sparky(5,"Sparky");
202:     cout << endl << sparky.GetName().GetString() << " is ";
203:     cout << sparky.GetAge() << " years old.";
204:     sparky.SetAge(8);
205:     cout << endl << sparky.GetName().GetString() << " is ";
206:     cout << sparky.GetAge() << " years old.";
207:     return 0;
208: }

```

### ▼ 输出:

```

String Tested OK String Tested OK String Tested OK String Tested OK String Teste
d OK String Tested OK String Tested OK String Tested OK String Tested OK String
Tested OK String Tested OK String Tested OK String Tested OK String Tested OK An
imal Tested OK String Tested OK Animal Tested OK
Sparky is Animal Tested OK 5 years old.Animal Tested OK Animal Tested OK Animal
Tested OK
Sparky is Animal Tested OK 8 years old.String Tested OK

```

### ▼ 分析:

在第 9~15 行定义了 ASSERT() 宏。如果定义了 DEBUG, 则将 ASSERT() 定义为在参数为 false 时打印一条错误消息。

第 39 行声明了 String 类成员函数 Invariants(), 其实现位于第 143~150 行定义。第 49~55 行声明了构造函数, 在对象构造好后, 第 54 行调用 Invariants() 来确认是否被正确构造。

对于其他构造函数重复该模式; 析构函数在它要开始毁坏对象前调用 Invariants(), 其他类函数在执行操作前调用 Invariants(), 并在返回前再次调用 Invariants()。这确认和证实了 C++ 的一个基本原则: 除构造函数和析构函数外, 所有成员函数都应作用于有效的对象, 并保持对象的有效状态。

第 176 行声明了 Animal 类的 Invariants() 方法, 该方法是在第 189~195 行实现的。注意, 在第 155、158、161 和 163 行, 内联函数可以调用 Invariants() 方法。

## 29.6.5 打印中间值

除使用 ASSERT() 宏确认事实为真外, 您可能还想打印指针、变量和字符串的当前值。这对于检验有关程序进程的假定以及确定 bug 在循环中的位置都很有帮助。程序清单 29.4 演示了如何操作。

程序清单 29.4 在调试模式下打印值

```

0: // Listing 29.4 - Printing values in DEBUG mode
1: #include <iostream>
2: using namespace std;
3: #define DEBUG
4:
5: #ifndef DEBUG
6:     #define PRINT(x)
7: #else
8:     #define PRINT(x) \
9:         cout << #x << ":\t" << x << endl;
10: #endif
11:
12: enum BOOL { FALSE, TRUE };
13:
14: int main()
15: {
16:     int x = 5;

```

```
17:    long y = 738981;
18:    PRINT(x);
19:    for (int i = 0; i < x; i++)
20:    {
21:        PRINT(i);
22:    }
23:
24:    PRINT (y);
25:    PRINT("Hi.");
26:    int *px = &x;
27:    PRINT(px);
28:    PRINT (*px);
29:    return 0;
30: }
```

▼ 输出:

```
x:      5
i:      0
i:      1
i:      2
i:      3
i:      4
y:      73898
"Hi.":  Hi.
px:      0012FEDC
*px:     5
```

▼ 分析:

第 6~9 行的 PRINT( )宏打印参数的当前值。注意，在第 9 行，传递给 cout 的第一个值是参数的字符串化版本，也就是说，如果传递 x，cout 将收到“x”。

接下来 cout 收到用引号括起的字符串“\t”，这将打印一个冒号和一个制表符。cout 收到的第 3 个值是参数 (x) 的值，最后收到的是 endl，它换行并刷新缓冲区。

注意，读者运行该程序时，显示的地址可能不是 0012FEDC。

29.7 位运算

经常需要在对象中设置标记以跟踪对象的状态：是否处于警告状态(AlarmState)？有没有初始化？要来还是要走？

为此，可以使用用户定义的布尔变量，但有些应用程序（尤其是低级驱动程序）和硬件设备要求您能够将变量的位用作标记。

每个字节包含 8 位，因此 4 字节的 long 变量能够存储 32 个标记。位的值为 1 时称为被设置，为 0 时称为被清除。设置位时使其值为 1；清除位时使其值为 0。可以通过修改 long 变量的值来设置或清除其位，但这样做很繁琐且容易令人迷惑。

注意

附录 A 提供了有关操纵二进制和十六进制数的宝贵信息。

C++提供的位运算符可以对变量的各个位进行操作。这些运算符看起来很像逻辑运算符，但实际上并不同，因此，很多新手常常弄混它们。表 29.1 列出了位运算符。

表 29.1 位运算符符号

符号	运算符	符号	运算符
&	与	^	异或
	或	~	求反



## 29.7.1 “与”运算符

“与”运算符为单个&，而逻辑“与”为两个&。对两个位进行“与”运算时，如果它们都是 1，则结果为 1；如果至少有一个为 0，则结果为 0。可以这样看待“与”运算：如果两个位都被设置，则结果为 1。

## 29.7.2 “或”运算符

第 2 个位运算符是“或”(|)。它用单个竖杠表示，而逻辑“或”用两个竖杠表示。对两个位进行“或”运算时，如果其中至少有个被设置，则结果为 1。

## 29.7.3 “异或”运算符

第 3 个位运算符是“异或”(^)。对两个位进行“异或”运算时，如果两个位的值相同（都为 0 或都为 1），则结果为 0。

## 29.7.4 “求反”运算符

“求反”运算符(~)清除被设置的位并设置被清除的位。如果当前值为 1010 0011，则“求反”结果为 0101 1100。

## 29.7.5 设置位

要设置或清除某个位，可使用掩码运算。如果有一个 4 字节的标记，要将第 8 位设置为 true，可对该标记和 128 执行“或”运算。

为什么呢？128 的二进制表示为 1000 0000。因此，第 8 位的位置值为 128。无论该位的当前值是什么（设置或清除），如果将其与值 128 执行“或”运算，都将设置该位而不改变其他位。假定变量的当前值为 1010011000100110，将其与 128 执行“或”运算时结果如下：

```
      9 8765 4321
1010 0110 0010 0110  // bit 8 is clear
| 0000 0000 1000 0000  // 128
-----
1010 0110 1010 0110  // bit 8 is set
```

读者应该发现了其他几点。首先，位通常从右向左编号。其次，在 128 的二进制表示中，除第 8 位（要设置的位）外，其他位皆为 0。最后，与 128 进行“或”运算后，原始值 1010 0110 0010 0110 除第 8 位被设置外，其他位不变。如果第 8 位已经被设置，它将保持被设置状态，这正是您希望的。

## 29.7.6 清除位

要清除第 8 位，可以将其同 128 的反码执行“与”运算。要得到 128 的反码，只需将 128 的位模式(10000000)中被设置的位清除，将被清除的位设置即可，结果为 01111111。将 128 的反码同变量执行“与”运算时，除第 8 位变为 0 外，变量的其他位不变。

```
1010 0110 1010 0110  // bit 8 is set
& 1111 1111 0111 1111  // ~128
-----
1010 0110 0010 0110  // bit 8 cleared
```

要理解这种解决方案，应自己执行运算。两个位都为1时，在结果中书写1；至少有一位为0时，在结果中书写0。将结果与原来值进行比较，将发现除第8位被清除外，其他位保持不变。

### 29.7.7 反转位

最后，要反转第8位（不管其当前状态如何），将变量同128执行“异或”运算。如果执行这种运算两次，结果将与原来的值相同。

```
1010 0110 1010 0110 // number
^ 0000 0000 1000 0000 // 128

1010 0110 0010 0110 // bit flipped
^ 0000 0000 1000 0000 // 128

1010 0110 1010 0110 // flipped back
```

#### 应该

应使用掩码和“或”运算符来设置位。  
应使用掩码和“与”运算符来清除位。  
应使用掩码和“异或”运算符来反转位。

#### 不应该

不要混淆各种位运算。  
反转位时，别忘了考虑它左边的位。一个字节包含8位，需要知道使用的变量有多少个字节。

### 29.7.8 位字段

在有些情况下，每个字节的空间都很宝贵，在类中节省6个和8个字节有天壤之别。如果类或结构中有系列布尔变量或只有少数几个可能取值的变量，可以使用位字段来节省内存空间。

在类中可以使用的最小的标准C++数据类型是char，它可能只占用1个字节。通常您会使用int类型，在使用32位处理器的计算机上，它占用4个字节。通过使用位字段，可以在char变量中存储8个二进制值，在4字节的int变量中存储32个二进制值。

位字段的命名和访问方式与其他类成员相同。它们的类型总是unsigned int，并在位字段名后加上冒号和数字。

这个数字告诉编译器，为该变量分配多少位内存。如果为1，位字段将能够表示0或1。如果为2，位字段将能够表示0、1、2或3，总共4个值。长3位的字段可表示8个值，依此类推。附录A简要地介绍了二进制数。程序清单29.5演示了位字段的用法。

程序清单 29.5 使用位字段

```
0: #include <iostream>
1: using namespace std;
2: #include <string.h>
3:
4: enum STATUS { FullTime, PartTime };
5: enum GRADLEVEL { UnderGrad, Grad };
6: enum HOUSING { Dorm, OffCampus };
7: enum FOODPLAN { OneMeal, AllMeals, WeekEnds, NoMeals };
8:
9: class student
10: {
11:     public:
12:         student():
13:             myStatus(FullTime),
14:             myGradLevel(UnderGrad),
15:             myHousing(Dorm),
16:             myFoodPlan(NoMeals)
17:     {}
```

```
18: ~student(){}
19: STATUS GetStatus();
20: void SetStatus(STATUS);
21: unsigned GetPlan() { return myFoodPlan; }
22:
23: private:
24:     unsigned myStatus : 1;
25:     unsigned myGradLevel: 1;
26:     unsigned myHousing : 1;
27:     unsigned myFoodPlan : 2;
28: };
29:
30: STATUS student::GetStatus()
31: {
32:     if (myStatus)
33:         return FullTime;
34:     else
35:         return PartTime;
36: }
37:
38: void student::SetStatus(STATUS theStatus)
39: {
40:     myStatus = theStatus;
41: }
42:
43: int main()
44: {
45:     student Jim;
46:
47:     if (Jim.GetStatus()== PartTime)
48:         cout << "Jim is part-time" << endl;
49:     else
50:         cout << "Jim is full-time" << endl;
51:
52:     Jim.SetStatus(PartTime);
53:
54:     if (Jim.GetStatus())
55:         cout << "Jim is part-time" << endl;
56:     else
57:         cout << "Jim is full-time" << endl;
58:
59:     cout << "Jim is on the " ;
60:
61:     char Plan[80];
62:     switch (Jim.GetPlan())
63:     {
64:         case OneMeal: strcpy(Plan,"One meal"); break;
65:         case AllMeals: strcpy(Plan,"All meals"); break;
66:         case WeekEnds: strcpy(Plan,"Weekend meals"); break;
67:         case NoMeals: strcpy(Plan,"No Meals");break;
68:         default :     cout << "Something bad went wrong! " << endl;
69:                     break;
70:     }
71:     cout << Plan << " food plan." << endl;
72:     return 0;
73: }
```

---

#### ▼ 输出:

```
Jim is part-time
Jim is full-time
Jim is on the No Meals food plan.
```

---

#### ▼ 分析:

第 4~7 行定义了几种枚举类型, 它们用于定义 student 类中位字段的可能取值。

第 9~28 行声明了 student 类。虽然这是一个很小的类, 但很有趣, 因为其所有数据都存储在 5 个位字段中 (第 24~27 行)。第 1 个位字段 (第 24 行) 表示学生的类别: 全职或业余。第 2 个位字段 (第 25 行) 表示学生是否为本科生。第 3 个位字段 (第 26 行) 表示学生是否住校。最后一个两位的位字

段表示 4 种就餐方式。

类方法的编写方式与其他类相同，不受数据移位字段而不是整型或枚举型的影响。

第 30~36 行的成员函数 `GetStatus()` 读取位字段的值，并返回一个枚举类型，但并非必须这样做，而可以直接返回位字段的值，编译器将进行转换。

为证明这一点，可以用下面的代码代替 `GetStatus()` 的实现：

```
STATUS student::GetStatus()
{
    return myStatus;
}
```

程序的功能不会有任何变化。这只是一个代码可读性问题，不影响编译器。

注意，第 47 行的代码必须检查类别，然后打印有意义的消息。如果写成下面这样：

```
cout << "Jim is " << Jim.GetStatus() << endl;
```

将打印：

```
Jim is 0
```

编译器不能将枚举常量 `PartTime` 转换为有意义的文本。

第 62 行使用 `switch` 语句判断就餐方式，并据此将一条相应的消息存储到字符数组 `plan` 中，然后在第 71 行打印该字符数组。也可以将 `switch` 语句写成这样：

```
case 0: strcpy(Plan, "One meal"); break;
case 1: strcpy(Plan, "All meals"); break;
case 2: strcpy(Plan, "Weekend meals"); break;
case 3: strcpy(Plan, "No Meals"); break;
```

使用位字段时最重要的一点是，类的客户无需关心数据存储实现。由于位字段被声明为私有的，因此以后可以修改它们，而无需修改接口。

## 29.8 编程风格

正如在本书其他地方指出的，采用一致的编码风格很重要，虽然从很多方面说，采用哪种编码风格无关紧要。一致的编程风格让人更容易猜测某部分代码的意图，还可避免查看上次调用函数时，首字母是否大写。

下面是一些指导原则，它们基于已完成的项目中使用的指导原则，事实证明它们很管用。您也可以制定自己的指导原则，但这些可供您开始时参考。

Emerson 说，“愚蠢的一致是一些零碎想法的混杂”，但在代码中保持一致是好事。制定自己的指导原则，但像它是由编程高手制定的一样对待它。

### 29.8.1 缩进

如果使用制表符，它应为 3 个空格。确保编辑器将制表符转换为 3 个空格。

### 29.8.2 大括号

如果对齐大括号是 C++ 程序员之间争论最为激烈的话题，下面推荐的一些做法：

- 匹配的大括号应垂直对齐；
- 定义和声明中最外面的大括号应在最左边，内部的语句应该缩进，其他所有大括号应与相关联的命令左对齐；

- 大括号应单独占一行。例如：

```
if (condition==true)
{
    j = k;
    SomeFunction();
}
m++;
```

**注意**

正如前面指出的，程序员们对大括号的对齐方式存在争议。很多 C++ 程序员认为，应将左大括号与相关联的命令放在同一行，并将右大括号与相关联的命令左对齐，如下所示：

```
if (condition==true) {
    j = k;
    SomeFunction();
}
```

这种格式被认为不方便阅读，因为大括号没有对齐。

### 29.8.3 长代码行和函数长度

确保不用水平滚动就能看到整行代码。超出右边界的代码易被忽略，且水平滚动很烦人。

将一行代码分成多行时，对后续行进行缩进。尽量在合理的地方分行，将插入运算符放在前一行的末尾（而不是下一行的开头），这样可清楚地知道，该行并不是完整的，后面还有其他代码。

在 C++ 中，函数通常比在 C 语言中短得多，但以前的合理建议仍然适用。尽量使函数足够短，以便可以在一页中打印函数的全部代码。

### 29.8.4 格式化 switch 语句

像下面这样缩进各个分支：

```
switch(variable)
{
    case ValueOne:
        ActionOne();
        break;
    case ValueTwo:
        ActionTwo();
        break;
    default:
        assert("bad Action");
        break;
}
```

正如读者看到的，稍微向右缩进了 case 语句。另外，对齐了每个分支中的语句。采用这种布局时，容易找到 case 语句及其后面的代码。

### 29.8.5 程序文本

可使用多种技巧来使代码易于阅读。易读的代码通常更容易维护。

- 使用空白来提高可读性。
- 不要在对象、数组名和运算符（.、->、[]）之间使用空格。
- 单目运算符与其操作数相关联，因此不要在它们之间添加空格，但在操作数的另一边添加空格。

单目运算符包括!、~、++、--、-、\*（用于指针）、&（取地址）和 sizeof。

- 双目运算符两边都应有空格，双目运算符包括+、=、\*、/、%、>>、<<、<、>、==、!=、&、|、&&、||、?:、+= 等。

- 不通过省略空格来指示优先级，如(4+ 3\*2)。
- 在逗号和分号后面加上空格，但在它们前面不加。
- 圆括号两边不应有空格。
- 用空格将关键字（如 if）分开，如 if(a == b)。
- 使用空格将单行注释的内容同//分开。
- 将指针或引用指示符紧靠类型名，而不是变量名，如下所示：

```
char* foo;
int& theInt;
```

```
rather than
char *foo;
int &theInt;
```

- 不在一行中声明多个变量。

## 29.8.6 标识符命名

下面是一些给标识符命名的指导原则。

- 标识符名称应足够长以便具有描述性。
- 避免意义不明确的缩写。
- 花时间和精力将含义拼写出来。
- 不使用匈牙利表示法。C++是强类型的，没有必要在变量名中包含类型名。对于用户定义的类型（类），匈牙利表示根本不管用。例外的情况是，在指针名中使用前缀 p、在引用名中使用前缀 r 以及在类成员变量名中使用前缀 its。
- 仅当简短性可提高代码的可读性或用途十分明显不需要描述性名称时，才使用短名称（i、p、x 等）。然而，通常应避免使用这样的名称。另外，应避免在变量名中使用字母 i、l 和 o，因为它们容易与数字混淆。
- 变量名的长度应与其作用域相称。
- 确保标识符看上去和听上去都相互不同，以尽可能减少混淆。
- 函数（或方法）名通常为动词或动词-名词短语，如 Secrch( )、Reset( )、FindParagraph( )、ShowCursor( )。变量名通常为抽象名词，可以带一个附加名词，如 count、state、windSpeed、windowHeight。布尔变量应相应地命名，如 windowIconized、fileIsOpen。

## 29.8.7 名称的拼写和大写

制定自己的编程风格时，不要忽略拼写和大写。下面是这方面的一些技巧。

- 对于使用#define 定义的常量，采用全部大写并用下划线将其中的单词分开，如 SOURCE\_FILE\_TEMPLATE。然而，在 C++ 中很少采用这种方式来定义常量，大多数情况下可考虑使用 const 和模板。
- 其他标识符应采用大小写混合，没有下划线。函数、方法、类、typedef 和结构的名称应采用首字母大写。诸如数据成员和局部变量等不应采用首字母大写。
- 枚举常量应以表示枚举类型缩写的小写字母开头，如：

```
enum TextStyle
{
    tsPlain,
    tsBold,
    tsItalic,
    tsUnderscore,
};
```

## 29.8.8 注释

注释可使程序更容易理解。有时候，编写程序期间可能暂停几天甚至几个月。在此期间，您可能忘记某些代码的功能或不知道为什么使用这些代码。别人阅读您的代码时，也可能无法理解。使用一致、深思熟虑的注释风格是值得的。请记住下面几条关于注释的技巧。

- 尽可能使用 C++ 单行注释（//）而不是多行注释（/\* \*/）。将多行注释用于将可能包含单行注释的代码块注释掉。
- 高级注释比处理细节更重要。要增加价值而不要复述代码，下面的注释根本不值得花时间去输入：n++; // n is incremented by one。将重点放在函数和代码块的语义上，指出函数的功能、副作用、参数类型和返回值，描述做出（或没有做出）的所有假设，如“假设 n 非负”或“如果 x 非法将



返回-1”。在复杂的逻辑中，使用注释来指出当前的状态。

- 使用采用合适标点符号和大小写的完整句子。这种额外的输入是值得的。注释不要过于晦涩，也不要使用缩写。编写代码时显而易见的事情几个月后将变得极其难懂。
- 使用空行来帮助读者理解所发生的事情，将语句分成逻辑组。

### 29.8.9 设置访问权限

访问程序各部分的方式应保持一致。下面是一些设置访问权限的技巧。

- 总是使用 `public:`、`private:`和 `protected:`，不要依赖于默认访问设置。
- 先列出公有成员，其次是保护成员，然后是私有成员。在方法后面集中列出数据成员。
- 在各个部分首先列出构造函数，然后是析构函数。集中列出同名的重载方法。尽可能集中列出存取器函数。
- 考虑按字母顺序排列每组中的方法和成员变量。包含文件时，按字母顺序排列它们。
- 虽然覆盖函数时关键字 `virtual` 是可选的，但应尽可能使用它。它有助于提醒函数虚函数以及保持声明的一致性。

### 29.8.10 类定义

尽可能确保方法实现的排列顺序与声明顺序一致，这可使方法更容易找到。

定义函数时，将返回值类型和其他所有限定符都放在前一行，让类名和函数名位于行首，这样更容易查找函数。

### 29.8.11 包含文件

尽可能少用 `#include`，最大限度地减少在头文件中包含的文件。理想情况是，只包含基类的头文件，其他必须包含的文件是声明类成员对象的头文件。

不要因为相应的 `.cpp` 文件需要包含某个文件，就在头文件中也包含该文件。不要因为被包含的文件需要包含某个文件而再包含该文件。

**提示**

所有头文件都应使用多重包含防范。

### 29.8.12 使用 `assert()`

本章前面介绍了 `assert()`。尽可能使用 `assert()`，它有助于发现错误，还可帮助了解程序编写者所做的假设以及他认为什么是合法的、什么是非法的。

### 29.8.13 使用 `const`

在合适的地方使用 `const`：参数、变量和方法。通常，方法需要有 `const` 版本和非 `const` 版本，不要以此为借口遗漏其中之一。显式地在 `const` 和非 `const` 之间进行转换时（有时，这是解决某些问题的唯一方式）务必小心，要确保这种转换有意义并进行注释。

## 29.9 C++开发工作的下一步

经过漫长而艰辛 C++ 学习，您具备了成为合格的 C++ 程序员所需的基本知识，但学习并没有结束。要从 C++ 新手成为 C++ 专家，还有很多的东西需要学习，需要从其他的地方获得有价值的信息。

接下来的几节推荐了很多信息源，这些推荐只反映笔者个人的经验和观点。有关这些主题的书籍和文章很多，购买之前一定要听听其他的观点。

### 29.9.1 从何处获得帮助和建议

作为一名 C++ 程序员，要做的第一件事就是加入 Internet 上的一个或多个 C++ 社区。这些小组让您能够及时地与数百甚至数千名 C++ 程序员取得联系，他们可以回答您的问题、提供建议提供以及发表看法的途径。

此外，您可能想寻找当地的 C++ 用户小组。很多城市都有 C++ 兴趣小组，在那里可以遇到其他程序员，同它们交流看法。

最后，诸如 Borland 和微软等编译器厂商也有新闻组，这些新闻组提供了有关开发环境和 C++ 语言的宝贵信息。

### 29.9.2 相关的 C++ 主题：托管 C++、C# 和微软的 .NET

微软 .NET 平台极大地改变了 Internet 开发方法，.NET 的一个重要组成部分是新语言 C# 和大量被称为托管扩展 (Managed Extension) 的 C++ 扩展。

C# 是 C++ 的扩展，为 C++ 编程人员转向 .NET 平台提供了桥梁。作为一种编程语言，C# 有一些不同于 C++ 的地方。例如，C# 不支持多继承，但使用接口提供了类似的功能。另外，C# 不使用指针，这消除了悬浮指针等问题，但代价是降低了该语言的低级、实时编程功能。有关 C# 需要指出的一点是，它使用了一个运行阶段库和无用单元收集程序 (GC)。GC 负责在必要时释放资源，避免了程序员这样做。

托管 C++ 也来自微软，它是 .NET 的组成部分。简单地说，这是一个 C++ 扩展，让 C++ 能够使用包括无用单元收集程序在内的所有 .NET 特性。

应该	不应该
<p>应阅读其他书籍。要学的知识很多，一本书不可能教给您需要的所有知识。</p> <p>应加入一个优秀的 C++ 用户组。</p>	<p>不要只阅读代码！学习 C++ 的最佳方法是编写 C++ 程序。</p>

## 29.10 总结

本章介绍了有关使用预处理器的更多细节。每次运行编译器时，预处理器都将首先运行，对诸如 #define 和 #ifdef 等预处理指令进行转换。

预处理器进行文本替换，虽然使用宏时，这种转换有点复杂。通过使用 #ifdef、#else 和 #ifndef，可以实现条件编译：在一组条件下编译一些语句，在另一组条件下编译另一些语句。这有助于编写用于多个平台的程序，常常用来有条件地包含调试信息。

宏函数根据编译时传递给宏的参数，提供复杂的文本替换。应用括号将宏的每个参数括起以确保进行正确的替换，这很重要。

相当于 C 语言，宏和预处理指令在 C++ 中不那么重要。C++ 提供了诸如 const 变量和模板等语言特性，它们可替代预处理指令且更高级。

本章还介绍了如何设置和测试位以及如何给类成员分配有限数目的位。

最后，讨论了 C++ 编程风格，提供了进一步学习的资源。

## 29.11 问与答

问：既然 C++ 提供了比预处理器更好的替代方案，为什么仍保留这种选项呢？

答：首先 C++ 向后与 C 语言兼容，C++ 必须支持 C 语言中所有的重要部分；其次，预处理器的某些用法在 C++ 中仍被频繁使用，如多重包含防范。

问：为什么在可以使用常规函数时仍使用宏函数？

答：宏函数以内联方式被展开，用于避免重复输入变化很小的命令。然而，模板提供一个更好的解决方案。

问：什么时候使用宏？什么时候使用内联函数？

答：尽可能使用内联函数。虽然宏提供字符替换、字符串化和字符串拼接功能，但它们不是类型安全的，可能导致代码更难以维护。

问：可使用什么来代替编译指令在调试期间打印中间值？

答：最好的替代方案是在调试器中使用 watch 语句。有关 watch 语句的信息，请参阅编译器或调试器文档。

问：何时该使用 assert()？何时该引发异常？

答：如果测试的条件在没有编程错误的情况下可以为真，则使用异常。如果仅当程序有错误时该条件才为真，则使用 assert()。

问：什么时候应使用位字段而不是 int 变量？

答：在对象的大小很重要时。如果可用的内存有限或编写的是通信软件，将发现使用位字段来节省空间对产品的成功至关重要。

问：可以将指针赋给位字段吗？

答：不可以。内存地址通常指向字节的开头，而位字段可能位于字节中间。

问：为什么有关风格的争论如此激烈？

答：程序员的习惯是根深蒂固的。如果您习惯了下面的缩进方式：

```
if (SomeCondition){  
    // statements  
}    // closing brace
```

改变起来将非常困难，总是认为新的编程风格不对，容易带来混乱。可以试着登录到某个流行的在线服务，问一下哪种缩进风格最好，哪一种编辑器最适合 C++，哪种产品是最好的字处理器，将看到上万条回信，所有回信都相互矛盾。

问：C++ 内容就这么多吗？

答：是的！您已学会了 C++，但总是有更多的知识需要学习。10 年前，学习所有关于某种计算机编程语言的知识是可能的，至少可以非常接近这种程度，而今天这是不可能的。即使尽最大努力，也不可能跟上潮流，行业在不断变化。务必不断阅读，经常查阅资源（杂志和在线服务），这样才能了解最新的变化。

## 29.12 作业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案，继续学习下一章前，请务必弄懂这些答案。

### 29.12.1 测验

1. 为何要使用断言？
2. 诸如 `__FILE__` 等预定义宏有何用途？
3. 在 2 字节的变量中可以存储多少位值？
4. 5 位可以存储多少个可能的值？
5. `0011 1100 | 1111 1111` 的结果是什么？
6. `0011 1100 & 1111 1111` 结果是什么？

### 29.12.2 练习

1. 编写防范多重包含头文件 `STRINGH` 的语句。
2. 编写一个 `assert()` 宏。如果调试级别是 2，它打印一条错误消息、文件名和行数；如果调试级别为 1，只打印一条消息（没有文件名和行数）；如果调试级别为 0，则什么都不打印。
3. 编写一个 `DPrint` 宏，它检测 `DEBUG` 是否被定义，如果被定义，则打印作为参数传入的值。
4. 编写一个类声明，这个类使用位字段来存储月份、年份和日。





## 附录 A

# 二进制和十六进制

读者很久以前就学过了基本的算术知识，如果没有这些知识后果将难以想象。看到 145 后，您不假思索就知道这是一百四十五。

一般通过以十进制格式来考虑数字，然而还可以使用其他格式来计数。使用计算机时，最常见的两种计数方式是二进制和十六进制。要理解二进制和十六进制，需要重新审视 145，将其视为一个数的代码，而不是数。

先从较小的开始：考虑数字 3 和“3”之间的关系，数字符号“3”是写在纸上的一个符号，而数字 3 是一个概念。数字符号用来表示数字。符号三、3、Ⅲ、III、\*\*\*都可用于表示数字 3 这个概念，由此可以清楚地看出符号和数字概念之间的区别。

在十进制中，使用 10 个符号（0、1、2、3、4、5、6、7、8、9）来表示所有的数。数字 10 如何表示呢？

可以设想使用字母 A 或 IIIIIIII 来表示 10。罗马人使用 X 来表示 10。在我们使用的阿拉伯计数方法中，结合使用位置和数字符号来表示值。第一位（最右边的那位）为个位，第二位为十位。因此数字十五表示为 15，即 1 个 10 和 5 个 1。

归纳起来，可以得到如下规则：

1. 十进制使用 10 个数字符号：0~9；
2. 各位表示 10 的幂（1、10、100）的倍数；
3. 最大的两位数为 99，推而广之，最大的  $n$  位数为  $10^n - 1$ 。因此，最大的三位数为  $10^3 - 1$  (999)。

### A.1 其他进制

人们使用十进制不是偶然的，这是因为人类有十根手指。然而，也可以使用其他进制。根据十进制中的规则，可以这样描述八进制：

1. 八进制使用 8 个符号：数字 0~7。
2. 八进制各位是 8 的幂：1、8、64 等。
3. 最大的  $n$  位八进制数为  $8^n - 1$ 。

为区分用不同进制表示的数，可将进制作为下标放在数的后面。十进制数十五可写作  $15_{10}$ 。

因此，要用八进制表示  $15_{10}$ ，可写作  $17_8$ 。注意它也可以读作“十五”因为那仍然是它所表示的数。

为什么是 17 呢？1 表示 1 个 8，7 表示 7 个 1，1 个 8 加 7 个 1 等于 15。请看下面 15 个星号：

\*\*\*\*\*

自然的做法是将它们分为两组，一组 10 个，另一组 5 个。这可用十进制表示为 15（1 个 10 和 5 个 1）。也可以这样将星号分为两组：

\*\*\*\*\*

即每组分别包含 8 个和 7 个。这可以用八进制表示为  $17_8$ ，即是 1 个 8 和 7 个 1。

## A.2 不同进制之间的转换

十五这个数可以用十进制表示为 15，用九进制表示为  $16_9$ ，用八进制表示为  $17_8$ ，用七进制表示为  $21_7$ 。为什么是  $21_7$  呢？因为七进制不使用数字符号 8，为表示十五，需要用 2 个 7 和 1 个 1。

如何归纳上述过程呢？要将十进制数转换为七进制，需考虑每位表示的值：在七进制中，它们是 1、7、49、343 等。为什么是这些值呢？因为它们分别是  $7^0$ 、 $7^1$ 、 $7^2$ 、 $7^3$  等。

任何数的 0 次幂（如  $7^0$ ）都是 1，1 次幂（如  $7^1$ ）为这个数本身，2 次幂是该数乘以本身（ $7^2 = 7 \times 7 = 49$ ），3 次幂是该数乘以本身再乘以本身（ $7^3 = 7 \times 7 \times 7 = 343$ ）。

可以创建一个这样的表：

位	4	3	2	1
幂	$7^3$	$7^2$	$7^1$	$7^0$
值	343	49	7	1

第一行为位号，第二行为 7 的幂，第三行为 7 的幂的十进制值。

将十进制数转换为七进制的步骤如下：检查十进制数，决定需要使用多少位才能表示它。例如，如果十进制数为 200，则只需使用三位，因此第 4 位（343）为 0，不必考虑。

为确定有多少个 49，用 49 去除 200。结果为 4，因此第 3 位为 4。余数为 4，它小于 7，因此第 2 位为 0。4 由 4 个 1 组成，因此第 1 位为 4。转换结果为  $404_7$ 。

位	4	3	2	1
幂	$7^3$	$7^2$	$7^1$	$7^0$
值	343	49	7	1
200 的七进制表示	0	4	0	4
十进制值	0	$4 \times 49 = 196$	0	$4 \times 1 = 4$

在这个例子中，第 3 位的 4 表示十进制值 196，第 1 位的 4 表示十进制值 4。 $196 + 4 = 200$ ，因此  $404_7 = 200_{10}$ 。

再来看一个例子，将十进制数 968 转换为六进制。在六进制中，各位表示的值如下：

位	5	4	3	2	1
幂	$6^4$	$6^3$	$6^2$	$6^1$	$6^0$
值	1296	216	36	6	1

为将十进制数 968 转换为六进制，从第 5 位开始。968 大于 1296 吗？否，因此第 5 位为 0。968 除以 216 的商为 4，余数为 104，因此第 4 位为 4。104 除以 36 的商为 2，余数为 32，因此第 3 位为 2。32 除以 6 的商为 5，余数为 2。因此结果为  $4252_6$ 。

位	5	4	3	2	1
幂	$6^4$	$6^3$	$6^2$	$6^1$	$6^0$
值	1296	216	36	6	1
968 的六进制表示	0	4	2	5	2
十进制值	0	$4 \times 216 = 864$	$2 \times 36 = 72$	$5 \times 6 = 30$	$2 \times 1 = 2$

### A.2.1 二进制

二进制是这种概念的终极扩展，它只使用两个符号：0 和 1。各位表示的值如下：

位	8	7	6	5	4	3	2	1
幂	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
值	128	64	32	16	8	4	2	1



要将十进制数 88 转换为二进制，可采取同样的步骤：88 小于 128，因此第 8 位为 0。

88 除以 64 的商为 1，余数为 24，因此第 7 位为 1。24 小于 32，因此第 6 位为 0。

24 除以 16 的商为 1，余数为 8，因此第 5 位为 1。8 除以 8 的商为 1，余数为 0，因此第 4 位为 1，其他位全为 0。

位	8	7	6	5	4	3	2	1
幂	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
值	128	64	32	16	8	4	2	1
88 的二进制表示	0	1	0	1	1	0	0	0
十进制值	0	64	0	16	8	0	0	0

为检验结果是否正确，对其做如下计算：

1 \* 64 = 64  
0 \* 32 = 0  
1 \* 16 = 16  
1 \* 8 = 8  
0 \* 4 = 0  
0 \* 2 = 0  
0 \* 1 = 0  
88

### A.2.2 为什么使用二进制

二进制与计算机要表示的东西相对应，因此二进制在编程中非常重要。计算机实际上根本不知道什么字母、数字符号、指令或程序。计算机的内核是电路，在某个给定的结点要么电压很高，要么很低。

为使逻辑清晰，工程师不用相对量表示电压，而是用两个量来表示：电压足够高和电压不够，他们不说“够”和“不够”，而是将其简化为“是”和“否”。“是”和“否”（或“真”和“假”）可以用 1 和 0 表示。按照约定，1 表示“是”（“真”），但这只是一种约定，也可以约定 1 表示“否”（“假”）。

有了这种认识上的飞跃后，二进制的威力就非常明显了：使用 1 和 0，可以表示每一段电路的真实状态（通电或断电）。计算机只知道“是”和“否”，“是”用 1 表示，“否”用 0 表示。

### A.2.3 位、字节和半字节

决定用 1 和 0 表示真和假后，二进制位就变得非常重要。由于早期的计算机每次发送 8 位，因此很自然地用 8 位数字来编写代码，8 位叫一个字节。

注意4 位被称为半字节。

8 个二进制位可以表示 256 个不同的值。为什么呢？如果 8 位都为 1，则值为 255 (128 + 64 + 32 + 16 + 8 + 4 + 1)；如果 8 位都为 0，则值为 0。0~255 有 256 种可能的状态。

### A.2.4 什么是 KB

$2^{10}$  (1024) 近似等于  $10^3$  (1000)。这种巧合有助于记忆，因此计算机科学家基于千的前缀为 kilo，将  $2^{10}$  个字节称为 1KB。

同样，由于  $1024 \times 1024$  (1048576) 接近与 1 百万，因此被称为 1MB，1024MB 被称为 1GB（十亿字节）。最后，1024G 被称为 1TB（万亿字节）。

A.2.5 二进制数

计算机用 1 和 0 对其要执行的任何操作进行编码。机器指令被编码成一系列的 1 和 0，由电路进行解释。任意一系列 1 和 0 都可以由计算机科学家翻译为数字，但认为这些数字有内在含义是错误的。

例如，Intel 8086 芯片组将位模式 1001 0101 解释为一条指令。您当然可以将其解释为十进制数 149，但是这个数本身是没有意义的。

有时，这些数为指令，有时为值，有时为编码。一种重要的标准编码集为 ASCII。在 ASCII 编码集中，每个字母和标点符号都用一个 7 位的二进制数表示。例如，小写字母 a 用 01100001 表示。这不是一个数，虽然可以将其解释为十进制数字 97 (64 + 32 + 1)。因此，人们说字母 a 用 ASCII 码 97 表示；实际情况是，97 的二进制表示 0110 0001 是字母 a 的编码，而十进制值 97 适合人类使用。

A.3 十六进制

由于二进制数难以阅读，于是人们想出了一种更为简单的表示方式。将二进制转换为十进制涉及大量的计算，但从二进制转换为十六进制非常简单，因为有一种非常好的快捷方法。

为理解这一点，必须先理解十六进制。十六进制使用十六个符号：0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F。最后六个是任选的，选择 A~F 是因为它们易于在键盘上表示。在十六进制中，各位表示的值如下：

位	4	3	2	1
幂	$16^3$	$16^2$	$16^1$	$16^0$
值	4 096	256	16	1

要将十六进制数转换为十进制，可以使用乘法。十六进制数 F8C 对应的十进制数为：

F \* 256 = 15 \* 256 = 3840  
8 \* 16 = 128  
C \* 1 = 12 \* 1 = 12  
3980

要将十六进制数 FC 转换为二进制，最好先将其转换为十进制，然后再转换为二进制：

F \* 16 = 15 \* 16 = 240  
C \* 1 = 12 \* 1 = 12  
252

将十进制数 252 转换为二进制需要使用如下表格：

位	9	8	7	6	5	4	3	2	1
幂	$2^8$	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
值	256	128	64	32	16	8	4	2	1

252 小于 256。因此第 9 位为 0；252 除以 128 的商为 1，余数为 124，因此第 8 位为 1；124 除以 64 的商为 1，余数为 60，因此第 7 位为 1；60 除以 32 的商为 1，余数为 28，因此第 6 位为 1；28 除以 16 的商为 1，余数为 12，因此第 5 位为 1；12 除以 8 的商为 1，余数为 4，因此第 4 位为 1；4 除以 4 的商为 1，余数为 0，因此第 3 位为 1，其他位全为 0。因此转换得到的二进制数为 1111 1100。

如果将该二进制数分成两组，每组 4 位，便可以将其奇妙地转换为十六进制。

右边一组为 1100，对应的十进制数为 12，十六进制数为 C。

左边一组为 1111，对应的十进制数为 15，十六进制数为 F。

因此 1111 1100 对应的十六进制数为 FC。这种快捷转换方法总是管用。可以将任何长度的二进制数划分成每 4 位一组，分别将每组转换为十六进制，再将这些十六进制数组合在一起，就得到了十六进制结果。下面是一个大得多的二进制数：

1011 0001 1101 0111

为验证上述快捷转换方法，首先将其转换为十进制。

在二进制中，每位表示的值依次翻倍。第 1 位为 1，第 2 位为 2，然后是 4、8、16，依此类推。

将上述二进制数转换为十进制的计算过程如下：

1×1	1
1×2	2
1×4	4
0×8	0
1×16	16
0×32	0
1×64	64
1×128	128
1×256	256
0×512	0
0×1 024	0
0×2 048	0
1×4 096	4 096
1×8 192	8 192
0×16 384	0
1×32 768	32 768
十进制值	45 527

要将这个十进制数转换为十六进制，需要使用如下表格：

位	5	4	3	2	1
幂	16 <sup>4</sup>	16 <sup>3</sup>	16 <sup>2</sup>	16 <sup>1</sup>	16 <sup>0</sup>
十进制值	65 536	4 096	256	16	1

45 527 小于 65 536，因此可以从第 4 位开始。45527 除以 4096 的商为 11，余数为 471；471 除以 256 的商为 1，余数为 215；215 除以 16 的商为 13，余数为 7。因此对应的十六进制数为 B1D7。

检验如下：

B (11) × 4096 =	45,056
1 × 256 =	256
D (13) × 16 =	208
7 × 1 =	7
总计	45,527

快捷方法是，将原来的二进制数 1011000111010111 分为 4 位一组：1011 0001 1101 0111。然后计算每组对应的十六进制数：

1011 =	
1 × 1 =	1
1 × 2 =	2
0 × 4 =	0
1 × 8 =	8
总计	11
十六进制位：	B

0001 =	
1 × 1 =	1
0 × 2 =	0
0 × 4 =	0
0 × 8 =	0
总计	1
十六进制位：	1



1101 =  
1 x 1 = 1  
0 x 2 = 0  
1 x 4 = 4  
1 x 8 = 8  
总计 13  
十六进制位: D

0111 =  
1 x 1 = 1  
1 x 2 = 2  
1 x 4 = 4  
0 x 8 = 0  
总计 7  
十六进制位: 7

十六进制数: B1D7

就像变魔术一样，从二进制转换为十六进制的简捷方法得到的结果与更复杂的方法相同。

在高级编程中，程序员非常频繁地使用十六进制，但是在很长的时间内，读者不使用十六进制也能高效地进行编程。

注意

一种使用十六进制的常见情形是处理颜色值。在 C++ 程序和其他领域（如 HTML）都如此。



## 附录 B

# C++关键字

关键字是语言保留给编译器使用的。不能将关键字用作类、变量或函数的名称。

asm	false	sizeof
auto	float	static
bool	for	static_cast
break	friend	struct
case	goto	switch
catch	if	template
char	inline	this
class	int	throw
const	long	true
const_cast	mutable	try
continue	namespace	typedef
default	new	typeid
delete	operator	typename
do	private	union
double	protected	unsigned
dynamic_cast	public	using
else	register	virtual
enum	reinterpret_cast	void
explicit	return	volatile
export	short	wchar_t
extern	signed	while

除上述关键字外，下述单词也被保留：

and	compl	or_eq
and_eq	not	xor
bitand	not_eq	xor_eq
bitor	or	

PDF

# 附录 C

## 运算符优先级

了解运算符有优先级很重要，但并不一定要记住运算符的优先级。

优先级是程序执行公式中运算的顺序。如果一个运算符的优先级高于另一个，则先执行它指定的计算。

优先级高的运算符比优先级低的运算符结合更紧密，因此，先执行优先级高的运算符指定的运算。在表 C.1 中，等级越低优先级越高。

表 C.1 运算符优先级

等级	名称	运算符
1	作用域解析运算符	::
2	成员选择、下标、函数调用、后缀递增和后缀递减	., ->, (), ++, --
3	sizeof、前缀递增和递减、求补、逻辑非、单目加和减、取址和解除引用、new、new[]、delete、delete[]、类型转换、sizeof()	++, --, ^, !, +, -, &, ()
4	用于指针的成员选择	.*, ->*
5	乘、除、求模	*, /, %
6	加、减	+, -
7	移位（左移和右移）	<<, >>
8	不等关系	<, <=, >, >=
9	相等关系	==, !=
10	按位“与”	&
11	按位“异或”	^
12	按位“或”	
13	逻辑“与”	&&
14	逻辑“或”	
15	条件运算符	?:
16	赋值运算符	=, *=, /=, %=, +=, -=, <<=, >>=, &=,  =, ^=
17	逗号运算符	,



# 附录 D

## 答 案

### 第 1 章

#### 测验

1. 解释器读取源代码并对程序进行翻译，将程序员的代码（程序指令）直接变成操作，编译器将源代码转换成可执行程序，该程序可在以后运行。
2. 随编译器而异。务必参阅编译器自带的文档。
3. 链接器的任务是将编译后的代码与编译器厂商和其他厂商提供的库链接起来。链接器让您能够分块地创建程序，然后将它们链接成一个大程序。
4. 编辑源代码、编译、连接、测试（运行），必要时重复上述步骤。

#### 练习

1. 该程序初始化两个 int 变量，然后打印它们的和（12）与积（35）。
2. 参见编译器手册。
3. 必须在第一行的 include 前加一个#。
4. 该程序将 Hello World 打印到控制台，然后另起一行（回车）。

### 第 2 章

#### 测验

1. 每次运行编译器时，预处理器都将首先运行。预处理器读取源代码，包含程序员要求包含的文件，执行其他辅助工作。然后编译器运行，将预处理后的源代码转换为目标代码。
2. 因为每当程序被执行时都将自动调用它。它可能不会被其他函数调用，但每个程序都必须有 main() 函数。
3. C++ 风格（单行）注释以两个斜杠（//）打头，将当前行尾之前的所有文本注释掉。C 风格（多行）注释由一对标记（/\*和\*/）指出，这对标记之间的所有内容都被注释掉。必须确保有匹配的标记对。
4. 可以在 C 风格（多行）注释内嵌套 C++ 风格（单行）注释，如下所示：  

```
/* This marker starts a comment. Everything including  
// this single line comment,  
is ignored as a comment until the end marker */
```

事实上，只要记住 C++ 风格注释到行尾结束，就可以将以 /\* 打头的注释嵌套在以 // 打头的 C++ 注释中。
5. C 风格（多行）注释可以。如果 C++ 风格注释超过一行，必须在每行注释开头加上一对斜杠（//）。

#### 练习

1. 下面是一种解决方案：

```
#include <iostream>
using namespace std;
int main()
{
    cout << "I love C++\n";
    return 0;
}
```

2. 下面的程序包含一个不执行任何操作的 main() 函数，然而，这是一个完整的程序，可以编译、链接和运行。当该程序运行时，好像什么事情也没有发生，因为它不执行任何操作。

```
int main(){}

```

3. 第 4 行的字符串左边少了一个双引号。

4. 下面是修正后的程序：

```
#include <iostream>
main()
{
    std::cout << "Is there a bug here?";
}
```

该程序将下述内容打印到屏幕上：

```
Is there a bug here?

```

5. 下面是一种解决方案：

```
#include <iostream>
int Add (int first, int second)
{
    std::cout << "Add(), received " << first << " and " << second << "\n";
    return (first + second);
}

int Subtract (int first, int second)
{
    std::cout << "Subtract(), received " << first << " and " << second << "\n";
    return (first - second);
}

int main()
{
    using std::cout;
    using std::cin;

    cout << "I'm in main()!\n";
    int a, b, c;
    cout << "Enter two numbers: ";
    cin >> a;
    cin >> b;

    cout << "\nCalling Add()\n";
    c=Add(a,b);
    cout << "\nBack in main().\n";
    cout << "c was set to " << c;
    cout << "\n\nCalling Subtract()\n";
    c=Subtract(a,b);
    cout << "\nBack in main().\n";
    cout << "c was set to " << c;

    cout << "\nExiting...\n\n";
    return 0;
}
```

## 第 3 章

### 测验

1. 整型变量是整数，浮点变量是实数，有“浮动”的小数点。浮点数可以用尾数和指数表示。
2. 关键字 `unsigned` 意味着 `int` 变量只能为正。在大多数使用 32 位处理器的计算机中，`short int` 占

用2字节，而 long int 占用4字节。然而，唯一的保证是，long int 不短于 int，而后者不短于 short int。通常，long int 的长度为 short int 的两倍。

3. 符号常量含义直观，常量的名称指出了其含义。另外，可在源代码的某个地方重新定义符号常量，而程序员必须修改使用字面量的所有代码。

4. const 变量的类型是确定的，编译器能够检查使用它们的方式是否正确。另外，经过预处理期处理后，它们仍然存在，因此可以在调试器中使用它们的名称。最重要的是。C++标准不再支持使用#define 来声明常量。

5. 好的变量名指出了变量的用途，糟糕的变量名没有提供任何信息。myAge 和 PeopleOnTheBus 都是好的变量名，而 xjk 和 prndl 可能没有什么用处。

6. BLUE=102

7.

- a. 好;
- b. 非法;
- c. 合法但糟糕;
- d. 好;
- e. 合法但糟糕。

## 练习

1. 合适的选择如下:

- a. unsigned short int
- b. unsigned long int 或 unsigned float
- c. unsigned double
- d. unsigned short int

2. 下面是一种答案:

- a. myAge
- b. backYardArea
- c. StarsInGalaxy
- d. averageRainFall

3. 声明如下:

```
const float PI = 3.14159;
```

4. 下面的代码声明并初始化这样的变量:

```
float myPi = PI;
```

## 第4章

### 测验

1. SomeArray[0]和 SomeArray[24]。

2. 为每一维提供一个下标。例如，SomeArray[2][3][2]是一个三维数组，第一维包含2个元素，第二维包含3个元素，第三维包含2个元素。

3. SomeArray[2][3][2] = { 0 };

4.  $10 \times 5 \times 20 = 1000$ 。

5. 该字符串包含16个字符: 读者看到的15个和末尾的一个空字符。

6. 空字符。

### 练习

1. 下面是一种可能的解决方案。数组名可以不同,但要存储 3×3 的棋盘,数组名后面必须是[3][3]。

```
int GameBoard[3][3];
```

2. `int GameBoard[3][3] = { {0,0,0},{0,0,0},{0,0,0} }`或`int GameBoard[3][3] = { 0 }`。

3. 下面是一种解决方案,这里使用了函数 `strcpy()` 和 `strlen()`。

```
#include <iostream>
#include <string.h>
using namespace std;

int main()
{
    char firstname[] = "Alfred";
    char middlename[] = "E";
    char lastname[] = "Numan";
    char fullname[80];
    int offset = 0;

    strcpy(fullname,firstname);
    offset = strlen(firstname);
    strcpy(fullname+offset," ");
    offset += 1;
    strcpy(fullname+offset,middlename);
    offset += strlen(middlename);
    strcpy(fullname+offset,". ");
    offset += 2;
    strcpy(fullname+offset,lastname);

    cout << firstname << "." << middlename << "."
         << lastname << endl;
    cout << "Fullname: " << fullname << endl;

    return 0;
}
```

4. 该数组的元素为 5×4,但代码初始化的却是 4×5。

5. 您的本意是 `i<5`,却写成了 `i<=5`。循环在 `i==5` 和 `j==4` 时仍将运行,但数组中并没有元素 `SomeArray[5][4]`。

## 第 5 章

### 测验

1. 表达式是任何返回一个值的语句。
2. `x=5+7` 是一个值为 12 的表达式。
3. `201 / 4` 的值为 50。
4. `201 % 4` 的值为 1。
5. `myAge` 为 41, `a` 为 39, `b` 为 41。
6. `8+2*3` 的值为 14。
7. 前者将 3 赋给 `x` 并返回 3 (被解释为 `true`),后者检测 `x` 是否等于 3,如果是则返回 `true`,否则返回 `false`。
8. 答案如下:
  - a. `false`
  - b. `true`
  - c. `true`
  - d. `false`

e. true

## 练习

1. 下面是一种解决方案:

```
if (x > y)
    x = y;
else      // y > x || y == x
    y = x;
```

2. 参见练习 3 的答案。

3. 输入 20、10 和 50 时, 结果 a 为 20, b 为 30, c 为 10。

第 14 行是赋值而不是检测是否相等。

4. 参见练习 5 的答案。

5. 第 6 行将 a-b 的值赋给 c, 而 a-b 为 0。由于 0 被视为 false, 因此 if 条件不满足, 不打印任何内容。

## 第 6 章

### 测验

1. 函数原型声明函数; 函数定义则定义函数。原型以分号结尾, 函数定义没有。声明中可以包括关键字 inline 和参数的默认值, 而定义不可以。声明中可以不包含参数的名称, 定义中必须包括。

2. 不, 所有参数都是根据位置而不是名称来识别的。

3. 将函数的返回类型声明为 void。

4. 未显式声明时, 函数的返回类型默认为 int。一种良好的编程习惯是, 总是明确地声明返回类型。

5. 局部变量是被传入代码块 (通常为函数) 或在代码块中声明的变量。局部变量仅在代码块中可见。

6. 作用域是指局部变量和全局变量的可见性与生存期, 作用域边界通常是由一组大括号指定。

7. 递归通常指函数调用自身的能力。

8. 全局变量通常在多个函数需要访问相同数据时使用。在 C++ 中很少用全局变量, 当您知道如何创建静态类变量后, 几乎不会创建全局变量。

9. 函数重载是指编写多个名称相同, 但参数数目或类型不同的函数的能力。

### 练习

1. unsigned long int Perimeter(unsigned short int, unsigned short int);

2. 下面是一种解决方案:

```
unsigned long int Perimeter(unsigned short int length,
                             unsigned short int width)
{
    return (2*length) + (2*width);
}
```

3. 该函数的返回类型被声明为 void, 因此不能返回值, 但却试图返回一个值。

4. 在该函数的定义中, 函数头末尾有一个分号, 如果不是这样, 它将是正确的。

5. 下面是一种解决方案:

```
short int Divider(unsigned short int valOne, unsigned short int valTwo)
{
    if (valTwo == 0)
        return -1;
    else
        return valOne / valTwo;
}
```

6. 下面是一种解决方案:

```

#include <iostream>
using namespace std;

short int Divider(
    unsigned short int valone,
    unsigned short int valtwo);

int main()
{
    unsigned short int one, two;
    short int answer;
    cout << "Enter two numbers.\n Number one: ";
    cin >> one;
    cout << "Number two: ";
    cin >> two;
    answer = Divider(one, two);
    if (answer > -1)
        cout << "Answer: " << answer;
    else
        cout << "Error, can't divide by zero!";
    return 0;
}

short int Divider(unsigned short int valOne, unsigned short int valTwo)
{
    if (valTwo == 0)
        return -1;
    else
        return valOne / valTwo;
}

```

#### 7. 下面是一种解决方案:

```

#include <iostream>
using namespace std;
typedef unsigned short USHORT;
typedef unsigned long ULONG;

ULONG GetPower(USHORT n, USHORT power);
int main()
{
    USHORT number, power;
    ULONG answer;
    cout << "Enter a number: ";
    cin >> number;
    cout << "To what power? ";
    cin >> power;
    answer = GetPower(number, power);
    cout << number << " to the " << power << "th power is " <<
        answer << endl;
    return 0;
}

ULONG GetPower(USHORT n, USHORT power)
{
    if (power == 1)
        return n;
    else
        return (n * GetPower(n, power-1));
}

```

## 第7章

### 测验

1. 用逗号将初始化分开, 如:  
for (x = 0, y = 10; x < 100; x++, y++).
2. goto 可以沿任何方向跳到任何一行代码, 这使源代码难以理解, 进而难以维护。
3. 可以。如果初始化后条件为 false, 则 for 循环体将不会执行。下面是一个例子:



```
for (int x = 100; x < 100; x++)
```

4. 变量 `x` 不在作用域中，因此没有有效的值。

5. 可以，任何循环都可嵌套在其他循环中。

6. 可以，下面是 `for` 循环和 `while` 循环的例子：

```
for(;;)
{
    // This for loop never ends!
}
while(true)
{
    // This while loop never ends!
}
```

7. 程序将好像被挂起，因为它永远不会结束。这导致您必须重新启动计算机或使用操作系统的高级特性来结束任务。

## 练习

1. 下面是一种解决方案：

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
        cout << "0";
    cout << endl;
}
```

2. 下面是一种解决方案：

```
for (int x = 100; x <= 200; x+=2)
```

3. 下面是一种解决方案：

```
int x = 100;
while (x <= 200)
    x+= 2;
```

4. 下面是一种解决方案：

```
int x = 100;
do
{
    x+=2;
} while (x <= 200);
```

5. `counter` 没有递增，`while` 循环永远不会结束。

6. 循环语句后有分号，因此该循环不执行任何操作。程序员的意图也许就是这样的，但如果本意是想通过 `counter` 打印每个值，将达不到目的，而只打印 `for` 循环结束后 `counter` 的值。

7. `counter` 被初始化为 100，但测试条件却为 `counter` 是否小于 10，因此不满足条件，循环体永远不会执行。如果将第 1 行改为 `int counter=5;`，循环将在数到最小的 `int` 变量取值时结束。`int` 默认为带符号的，这不是程序员的初衷。

8. `case 0` 可能需要一条 `break` 语句。如果没有，应通过注释加以说明。

## 第 8 章

### 测验

1. 地址运算符 (`&`)。

2. 解除引用运算符 (`*`)。

3. 指针是一个变量，它存储的是另一个变量的地址。

4. 指针存储的地址是另一个变量的地址。存在该地址中的值是多少变量的值。间接运算符 (`*`) 返回存储在指定地址中的值，而地址本身存储在指针中。

5. 间接运算符返回指针指向的地址中的值，地址运算符 (`&`) 返回变量的地址。

6. 前者将 ptrOne 声明为一个指向 int 常量的指针, 使用该指针不能修改 int 常量的值, 后者将 ptrTwo 声明为指向 int 变量的常量指针, 初始化该指针后便不能给它重新赋值。

### 练习

1.

a. `int *pOne;` 声明一个 int 指针

b. `int vTwo;` 声明一个 int 变量

c. `int * pThree=&vTwo;` 声明一个 int 指针并将其初始化为变量 vTwo 的地址。

2. `unsigned short *pAge = &yourAge;`

3. `*pAge = 50;`

4. 下面是一种解决方案:

```
#include <iostream>

int main()
{
    int theInteger;
    int *pInteger = &theInteger;
    *pInteger = 5;

    std::cout << "The Integer: "
               << *pInteger << std::endl;

    return 0;
}
```

5. 应初始化 pInt。更重要的是, 由于它没有被初始化: 没有将任何内存地址赋给它, 因此指向内存中的一个随机位置。将字面量 9 存储到这个随机位置中是一种非常危险的错误。

6. 程序员的意图可能是将 9 赋给 pVar 指向的变量 SomeVariable, 然而, 由于遗漏了间接运算符 (\*), 实际上却将 9 赋给了 pVar。如果使用 pVar 来赋值将导致灾难, 因为它指向地址 9, 而不是变量 SomeVariable。

## 第 9 章

### 测验

1. 引用是别名, 而指针是存储地址的变量。引用不能为空, 也不能给它赋值。
2. 需要重新赋值或指向的目标可能为空时。
3. 空指针 (0)。
4. 指向常量对象的引用的简称。
5. 按引用传递意味着不创建局部副本。可通过按引用或指针传递来实现。
6. 都正确, 但应选择其中一种并始终使用它。

### 练习

1. 下面是一种解决方案:

```
#include <iostream>

int main()
{
    int varOne = 1;        // sets varOne to 1
    int& rVar = varOne;
    int* pVar = &varOne;
    rVar = 5;              // sets varOne to 5
    *pVar = 7;             // sets varOne to 7
}
```

```
// All three of the following will print 7:
std::cout << "variable: " << varOne << std::endl;
std::cout << "reference: " << rVar << std::endl;
std::cout << "pointer: " << *pVar << std::endl;
```

```
return 0;
}
```

## 2. 下面是一种解决方案:

```
int main()
{
    int varOne;
    const int * const pVar = &varOne;
    varOne = 6;
    *pVar = 7;
    int varTwo;
    pVar = &varTwo;
    return 0;
}
```

## 3. 不能给常量对象赋值, 不能重新给常量指针赋值。这意味着第 6 行和第 8 行有问题。

## 4. 下面是一种答案。注意, 由于迷失指针, 运行该程序将是危险的。

```
int main()
{
    int * pVar;
    *pVar = 9;
    return 0;
}
```

## 5. 下面是一种答案:

```
int main()
{
    int varOne;
    int * pVar = &varOne;
    *pVar = 9;
    return 0;
}
```

## 6. 下面是一种答案。注意, 在程序中应避免内存泄漏。

```
#include <iostream>
int FuncOne();
int main()
{
    int localVar = FuncOne();
    std::cout << "The value of localVar is: " << localVar;
    return 0;
}

int FuncOne()
{
    int * pVar = new int (5);
    return *pVar;
}
```

## 7. 下面是一种解决方案:

```
#include <iostream>
void FuncOne();
int main()
{
    FuncOne();
    return 0;
}

void FuncOne()
{
    int * pVar = new int (5);
    std::cout << "The value of *pVar is: " << *pVar ;
    delete pVar;
}
```

8. MakeCat 返回一个指向自由存储区中创建的 CAT 对象的引用。没有办法释放这些内存, 这将导致内存泄漏。

9. 下面是一种解决方案:

```
#include <iostream>
using namespace std;
class CAT
{
public:
    CAT(int age) { itsAge = age; }
    ~CAT(){}
    int GetAge() const { return itsAge;}
private:
    int itsAge;
};

CAT * MakeCat(int age);
int main()
{
    int age = 7;
    CAT * Boots = MakeCat(age);
    cout << "Boots is " << Boots->GetAge() << " years old";
    delete Boots;
    return 0;
}

CAT * MakeCat(int age)
{
    return new CAT(age);
}
```

## 第 10 章

### 测验

1. 句点运算符是., 可用于访问类或结构的成员。
2. 变量定义分配内存; 类声明不分配内存。
3. 类声明是类的接口, 告诉类的客户如何与类交互。类实现是成员函数的定义, 通常位于相关的cpp文件中。
4. 公有数据成员可被类的客户直接访问, 私有数据成员只能被类的成员函数访问。
5. 可以。虽然本章没有指出, 但成员函数确实可以是私有的。只有类的其他成员函数才能调用私有成员函数。
6. 虽然成员数据可以是公有的, 但良好的编程习惯是将其声明为私有的, 并提供存取这些数据的公有存取器函数。
7. 可以, 每个类对象都有自己的数据成员。
8. 声明在右大括号后以分号结尾, 函数定义没有。
9. 在 Cat 类中, 不接受任何参数且返回类型为 void 的成员函数 Meow() 的函数头如下:  
void Cat::Meow()
10. 调用构造函数来初始化类。这个特殊函数的名称与类名相同。

### 练习

1. 下面是一种解决方案:

```
class Employee
{
    int Age;
    int YearsOfService;
    int Salary;
};
```

2. 下面是一种解决方案。注意存取器方法 Get... 也被声明为 const 的, 因为它们不对对象做任何修改。

```
// Employee.h
class Employee
{
public:
    int  GetAge() const;
    void SetAge(int age);
    int  GetYearsOfService() const;
    void SetYearsOfService(int years);
    int  GetSalary() const;
    void SetSalary(int salary);

private:
    int itsAge;
    int itsYearsOfService;
    int itsSalary;
};
```

### 3. 下面是一种解决方案:

```
// Employee.cpp
#include <iostream>
#include "Employee.h"

int Employee::GetAge() const
{
    return itsAge;
}
void Employee::SetAge(int age)
{
    itsAge = age;
}
int Employee::GetYearsOfService() const
{
    return itsYearsOfService;
}
void Employee::SetYearsOfService(int years)
{
    itsYearsOfService = years;
}
int Employee::GetSalary()const
{
    return itsSalary;
}
void Employee::SetSalary(int salary)
{
    itsSalary = salary;
}
int main()
{
    using namespace std;

    Employee John;
    Employee Sally;

    John.SetAge(30);
    John.SetYearsOfService(5);
    John.SetSalary(50000);

    Sally.SetAge(32);
    Sally.SetYearsOfService(8);
    Sally.SetSalary(40000);

    cout << "At AcmeSexist company, John and Sally have ";
    cout << "the same job.\n\n";

    cout << "John is " << John.GetAge() << " years old." << endl;
    cout << "John has been with the firm for ";
    cout << John.GetYearsOfService() << " years." << endl;
    cout << "John earns $" << John.GetSalary();
    cout << " dollars per year.\n\n";

    cout << "Sally, on the other hand is " << Sally.GetAge();
    cout << " years old and has been with the company ";
    cout << Sally.GetYearsOfService();
```

```

    cout << " years. Yet Sally only makes $" << Sally.GetSalary();
    cout << " dollars per year! Something here is unfair.";
}

```

4. 下面是正确答案之一:

```

float Employee::GetRoundedThousands() const
{
    return Salary / 1000;
}

```

5. 下面是一种解决方案:

```

class Employee
{
public:
    Employee(int age, int years, int salary);
    int GetAge() const;
    void SetAge(int age);
    int GetYearsOfService() const;
    void SetYearsOfService(int years);
    int GetSalary() const;
    void SetSalary(int salary);
private:
    int itsAge;
    int itsYearsOfService;
    int itsSalary;
};

```

6. 声明必须以分号结尾。

7. 存取器函数 GetAge() 是私有的。记住，除非特别声明，否则所有类成员都是私有的。

8. 不能直接访问 itsStation，因为它是私有的，不能对类调用 SetStation()，只能对对象调用 SetStation()；不能初始化 myOtherTV，因为没有匹配的构造函数。

## 第 11 章

### 测验

1. 虚函数表是 C++ 编译器管理虚函数的一种常用方式。该表记录了所有虚函数的地址，根据指向的对象的运行阶段类型调用正确的函数。

2. 任何类的析构函数都可声明为虚的。对指针调用 delete 时，将确定被指向对象的运行阶段类型，据此调用正确的派生类析构函数。

3. 这是一个伪问题：没有虚构造函数。

4. 在类中创建一个虚方法，该方法调用复制构造函数。

5. Base::FunctionName();

6. FunctionName();

7. 是的，虚拟性将被继承，不能撤销。

8. 派生类的成员函数可以访问基类的保护成员。

### 练习

1. virtual void SomeFunction(int);

2. 声明 Square 时，无需考虑 Shape。Shape 将作为 Rectangle 的一部分自动包含进来。

```

class Square : public Rectangle
{};

```

3. 与练习 2 一样，不用考虑 Shape。

```

Square::Square(int length):
    Rectangle(length, width){}

```

4. 下面是一种解决方案:



```
class Square
{
public:
    // ...
    virtual Square * clone() const { return new Square(*this); }
    // ...
};
```

5. 也许没有错。SomeFunction 需要一个 Shape 对象，已经传递给它一个从 Rectangle “切成”的 Shape。只要不需要 Rectangle 部分，这就是可行的。如果需要 Rectangle 部分，则必须修改 SomeFunction，使之接受一个 Shape 指针或 Shape 引用作为参数。

6. 不能将复制构造函数声明为虚函数。

## 第 12 章

### 测验

1. 向下转换是将基类指针视为派生类指针。

2. 这指的是将共有的功能向上移到共同基类中。如果多个类都有某个函数，应寻找它们共同的基类，将该函数放在该基类中。

3. 如果声明时没有使用关键字 virtual，将创建两个 Shapes：一个属于 Rectangle，另一个属于 Shape。如果声明类时在这两个类名前使用了关键字 virtual，将只创建一个 Shape。

4. Horse 和 Bird 都在其构造函数中初始化基类 Animal，Pegasus 也如此。创建 Pegasus 对象时，Horse 和 Bird 对 Animal 的初始化将被忽略。

5. 下面是一种答案：

```
class Vehicle
{
    virtual void Move() = 0;
};
```

6. 都不需要覆盖，除非要使类成为非抽象的，在这种情况下都必须覆盖。

### 练习

1. `class JetPlane : public Rocket, public Airplane`

2. `class Seven47: public JetPlane`

3. 下面是一种答案：

```
class Vehicle
{
    virtual void Move() = 0;
    virtual void Haul() = 0;
};
```

```
class Car : public Vehicle
{
    virtual void Move();
    virtual void Haul();
};
```

```
class Bus : public Vehicle
{
    virtual void Move();
    virtual void Haul();
};
```

4. 下面是一种答案：

```
class Vehicle
{
    virtual void Move() = 0;
    virtual void Haul() = 0;
};
```



```

class Car : public Vehicle
{
    virtual void Move();
};

class Bus : public Vehicle
{
    virtual void Move();
    virtual void Haul();
};

class SportsCar : public Car
{
    virtual void Haul();
};

class Coupe : public Car
{
    virtual void Haul();
};

```

## 第 13 章

### 测验

1. 可以。编译器使用 const 版本执行读取操作，并使用非 const 版本执行写入操作。程序员可使用这种功能定制读写行为。例如，在多线程环境中，读取可能是安全的，但需要同步写入操作。这样可能提高读取的性能。

2. 需要定义的运算符包括：赋值运算符使得能够编写代码 `strDest = strSource;`，运算符`+`用于拼接字符串，并使得能够编写代码 `strDest = strSource1 + strSource2;`，运算符`+=`用于在 string 对象后面追加字符串，使得能够编写代码 `strDest += strAdd;`，运算符`<`用于比较，使用 STL 排序函数对 string 对象进行排序时，这很有用，运算符`==`根据两个 string 对象的内容比较它们。还需定义其他运算符，如用于访问各个字符的下标运算符（`[]`）以及用于将字符串转换为常用类型的转换运算符。

3. 不需要。即使日期是以按位格式进行操作的，诸如`&`、`!`、`^`和`|`等按位运算符也没有意义，因为没有程序员会这样使用。

### 练习

1. 可这样声明单目递减运算符的前缀和后缀版本：

```

CDate& operator -- ()
CDate operator -- (int)

```

2. 为满足需求，按如下方式定义运算符`==`和`<`：

```

bool CMyArray::operator==(const CMyArray& dest) const
{
    bool bRet = false;
    if (dest.m_nNumElements == this->m_nNumElements)
    {
        for (int nIndex = 0; nIndex < m_nNumElements; ++ nIndex)
        {
            bRet = dest[nIndex] == m_pnInternalArray[nIndex];
            if (!bRet)
                break;
        }
    }
    return bRet;
}

bool CMyArray::operator<(const CMyArray& dest) const
{
    bool bRet = false;

    if (m_nNumElements < dest.m_nNumElements)

```

```

        bRet = true;
    else if (m_nNumElements > dest.m_nNumElements)
        bRet = false;
    else // equal lengths
    {
        int nSumArrayContents = 0;
        int nSumDestArrayContents = 0;

        for (int nIndex = 0; nIndex < m_nNumElements; ++ nIndex)
        {
            nSumArrayContents += m_pnInternalArray [nIndex];
            nSumDestArrayContents += dest.m_pnInternalArray [nIndex];
        }

        bRet = (nSumArrayContents < nSumDestArrayContents);
    }

    return bRet;
}

```

## 第 14 章

### 测验

1. dynamic\_cast
2. 当然是修改函数。一般而言，应将 const\_cast 和类型转换运算符作为最后的手段。
3. 对。
4. 对。

## 第 15 章

### 测验

1. 多重包含防范用于避免在程序中多次包含头文件。
2. #define debug 0 将 debug 定义为 0，每当在代码中发现 debug 时，都将把它替换为字 0。#undef debug 删除 debug 的定义，但不替换源文件中的 debug。
3. 答案为 4/2，即 2。
4. 结果为 10 + 10/2，即 10 + 5 或 15，这显然不是期望的结果。
5. 应加上括号：HALVE (x) ((x)/2)。
6. 模板是一种抽象，在编译阶段被实例化为具体形式，而宏更像替换字符串的常量。
7. 模板参数用于指定为哪种类型实例化模板，而函数参数指定函数可接受哪种类型的对象。
8. 标准模板库是大多数 C++ 开发环境的一个重要组成部分，它向程序员提供模板类和函数，以解决很多常见的计算问题，让程序员无需重新编写这些算法。

### 练习

1. 宏 ADD 的定义如下：  
#define ADD(x,y) (x + y)
2. 模板函数 ADD 的定义如下：  
template <typename T>  
T Add (const T& x, const T& y)  
{  
 return (x + y);  
}
3. 模板函数 swap 的定义如下：

```
template <typename T>
void Swap (T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

4. #define QUARTER(x) ((x)/ 4)

5. 该模板类的定义类似于下面这样:

```
template <typename Array1Type, typename Array2Type>
class CTwoArrays
{
private:
    Array1Type m_Array1 [10];
    Array2Type m_Array2 [10];
public:
    Array1Type& GetArray1Element(int nIndex){return m_Array1[nIndex];}
    Array2Type& GetArray2Element(int nIndex){return m_Array2[nIndex];}
};
```

## 第 16 章

### 测验

1. deque。只有 deque 支持容器开头和末尾插入元素，且时间是固定的。
2. 如果要存储的是键-值对，应选择 std::set 或 std::map；如果有重复的元素，应选择 std::multiset 或 std::multimap。
3. 可能。在实例化 std::set 模板时，可提供第二个模板参数，它是一个二元谓词，set 类将它用作排序标准。可根据应用程序的需求定义该二元谓词，它必须能够对元素进行排序。
4. 迭代器在算法和容器之间架设了桥梁，让算法能够在不知道容器类型的情况下对容器进行操作。
5. hash\_set 并非遵循 C++ 标准的容器，因此不应在有移植性需求的应用程序中使用它。在这种情况下，应使用 std::map。

## 第 17 章

### 测验

1. std::basic\_string<T>。
2. 将两个字符串复制到两个副本对象中，再将每个复制的字符串都转换为小写或大写。然后对转换后的字符串进行比较，并返回比较结果。
3. 否。C 风格字符串实际上是类似于字符数组的指针，而 STL string 是一个类，它实现了各种运算符和成员函数，使字符串操作和处理尽可能简单。

### 练习

1. 该程序需要使用 std::reverse:

```
#include <string>
#include <iostream>
#include <algorithm>

int main ()
{
    using namespace std;

    cout << "Please enter a word for palindrome-check:" << endl;
    string strInput;
```

```
cin >> strInput;

string strCopy (strInput);
reverse (strCopy.begin (), strCopy.end ());
if (strCopy == strInput)
    cout << strInput << " is a palindrome!" << endl;
else
    cout << strInput << " is not a palindrome." << endl;

return 0;
}

2. 使用 std::find:
#include <string>
#include <iostream>

using namespace std;

// Find the number of character 'chToFind' in string "strInput"
int GetNumCharacters (string& strInput, char chToFind)
{
    int nNumCharactersFound = 0;

    size_t nCharOffset = strInput.find (chToFind);
    while (nCharOffset != string::npos)
    {
        ++ nNumCharactersFound;

        nCharOffset = strInput.find (chToFind, nCharOffset + 1);
    }

    return nNumCharactersFound;
}

int main ()
{
    cout << "Please enter a string:" << endl << "> ";
    string strInput;
    getline (cin, strInput);

    int nNumVowels = GetNumCharacters (strInput, 'a');
    nNumVowels += GetNumCharacters (strInput, 'e');
    nNumVowels += GetNumCharacters (strInput, 'i');
    nNumVowels += GetNumCharacters (strInput, 'o');
    nNumVowels += GetNumCharacters (strInput, 'u');

    // DIY: handle capitals too..

    cout << "The number of vowels in that sentence is: " << nNumVowels;

    return 0;
}
```

### 3. 使用函数 toupper:

```
#include <string>
#include <iostream>
#include <algorithm>

int main ()
{
    using namespace std;

    cout << "Please enter a string for case-conversion:" << endl;
    cout << "> ";

    string strInput;
    getline (cin, strInput);
    cout << endl;

    for ( size_t nCharIndex = 0
        ; nCharIndex < strInput.length ()
        ; nCharIndex += 2)
```



```

        strInput[nCharIndex] = toupper(strInput[nCharIndex]);

    cout << "The string converted to upper case is: " << endl;
    cout << strInput << endl << endl;

    return 0;
}

```

4. 可以这样编写程序:

```

#include <string>
#include <iostream>

int main ()
{
    using namespace std;

    const string str1 = "I";
    const string str2 = "Love";
    const string str3 = "STL";
    const string str4 = "String.";

    string strResult = str1 + " " + str2 + " " + str3 + " " + str4;

    cout << "The sentence reads:" << endl;
    cout << strResult;

    return 0;
}

```

## 第 18 章

### 测验

1. 否。仅当在 vector 末尾插入元素时所需的时间才是固定的。
2. 10 个。插入第 11 个元素，将导致重新分配内存。
3. 删除最后一个元素，即删除末尾的元素。
4. 类型 CMammal。
5. 通过下标运算符（[]）或函数 at()。
6. 随机访问迭代器。

### 练习

1. 下面是一种解决方案:

```

#include <vector>
#include <iostream>

using namespace std;

char DisplayOptions ()
{
    cout << "What would you like to do?" << endl;
    cout << "Select 1: To enter an integer" << endl;
    cout << "Select 2: Query a value given an index" << endl;
    cout << "Select 3: To display the vector" << endl << "> ";
    cout << "Select 4: To quit!" << endl << "> ";

    char ch;
    cin >> ch;

    return ch;
}

int main ()
{
    vector<int> vecData;

```





```

char chUserChoice = '\0';
while ((chUserChoice = DisplayOptions ()) != '4')
{
    if (chUserChoice == '1')
    {
        cout << "Please enter an integer to be inserted: ";
        int nDataInput = 0;
        cin >> nDataInput;
        vecData.push_back (nDataInput);
    }
    else if (chUserChoice == '2')
    {
        cout << "Please enter an index between 0 and ";
        cout << (vecData.size () - 1) << ": ";
        int nIndex = 0;
        cin >> nIndex;

        if (nIndex < (vecData.size ()))
        {
            cout<<"Element ["<<nIndex<<"] = "<<vecData [nIndex];
            cout << endl;
        }
    }
    else if (chUserChoice == '3')
    {
        cout << "The contents of the vector are: ";
        for (size_t nIndex = 0; nIndex < vecData.size (); ++ nIndex)
            cout << vecData [nIndex] << ' ';
        cout << endl;
    }
}
return 0;
}

```

## 2. 使用 std::find 算法:

```

vector<int>::iterator iElementFound = std::find (vecData.begin (),
                                                vecData.end (), nDataInput);

```

## 3. 修改练习 1 的解决方案, 以接受用户输入并显示 vector 的内容。

# 第 19 章

## 测验

1. 可将元素插入到 list 中间, 可也将其插入到列表两端, 插入位置不会影响性能。
2. list 的独特之处在于, 这些操作不会导致现有的迭代器失效。
3. mList.clear ();或 mList.erase (mList.begin(), mList.end());
4. 可以。insert 函数的一个重载版本可用于插入集合中特定范围内的元素。

## 练习

1. 这类似于第 18 章中练习 1 的解决方案, 唯一需要修改的地方是使用 list 的 insert 函数, 如下所示:  
mList.insert (mList.begin(), nDataInput);
2. 存储两个指向 list 元素的迭代器, 使用 list 的 insert 函数在中间插入一个元素, 然后使用这两个迭代器来演示在插入元素后它们仍指向以前的元素。

## 3. 下面是一种可能的解决方案:

```

#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main ()
{

```

```

vector<int> vecData (4);
vecData [0] = 0;
vecData [1] = 10;
vecData [2] = 20;
vecData [3] = 30;

list<int> listIntegers;

// Insert the contents of the vector into the beginning of the list
listIntegers.insert (listIntegers.begin (),
                    vecData.begin (), vecData.end());

cout << "The contents of the list are: ";

list<int>::const_iterator iElement;
for ( iElement = listIntegers.begin ()
      ; iElement != listIntegers.end ()
      ; ++ iElement)
    cout << *iElement << " ";

return 0;
};

```

#### 4. 下面是一种可能的解决方案:

```

#include <list>
#include <string>
#include <iostream>

using namespace std;

int main ()
{
    list<string> listNames;
    listNames.push_back ("Jack");
    listNames.push_back ("John");
    listNames.push_back ("Anna");
    listNames.push_back ("Skate");

    cout << "The contents of the list are: ";

    list<string>::const_iterator iElement;
    for (iElement = listNames.begin(); iElement!=listNames.end();
        ++iElement)
        cout << *iElement << " ";
    cout << endl;

    cout << "The contents after reversing are: ";
    listNames.reverse ();
    for (iElement = listNames.begin(); iElement!=listNames.end();
        ++iElement)
        cout << *iElement << " ";
    cout << endl;

    cout << "The contents after sorting are: ";
    listNames.sort ();
    for (iElement = listNames.begin(); iElement!=listNames.end();
        ++iElement)
        cout << *iElement << " ";
    cout << endl;

    return 0;
};

```

## 第 20 章

### 测验

1. 默认排序标准由 `std::less` 指定, 它使用运算符 `<` 来比较两个整数, 并在第一个小于第二个时返回 `true`。
2. 重复的元素在一起, 它们彼此相邻。

3. size(), 所有 STL 容器都是。

## 练习

1. 该二元谓词可以是这样的:

```
struct FindContactGivenNumber
{
    bool operator()(const CContactItem& lsh, const CContactItem& rsh) const
    {
        return (lsh.strPhoneNumber < rsh.strPhoneNumber);
    }
};
```

2. 该结构和 multiset 的定义如下:

```
#include <set>
#include <iostream>
#include <string>

using namespace std;

struct PAIR_WORD_MEANING
{
    string strWord;
    string strMeaning;

    PAIR_WORD_MEANING (const string& sWord, const string& sMeaning)
        : strWord (sWord), strMeaning (sMeaning) {}

    bool operator< (const PAIR_WORD_MEANING& pairAnotherWord) const
    {
        return (strWord < pairAnotherWord.strWord);
    }
};

int main ()
{
    multiset <PAIR_WORD_MEANING> msetDictionary;
    PAIR_WORD_MEANING word1 ("C++", "A programming language");
    PAIR_WORD_MEANING word2 ("Programmer", "A geek!");

    msetDictionary.insert (word1);
    msetDictionary.insert (word2);

    return 0;
}
```

3. 下面是一种解决方案:

```
#include <set>
#include <iostream>

using namespace std;

template <typename T>
void DisplayContent (const T& sequence)
{
    T::const_iterator iElement;

    for (iElement = sequence.begin(); iElement != sequence.end(); ++iElement)
        cout << *iElement << " ";
}

int main ()
{
    multiset <int> msetIntegers;

    msetIntegers.insert (5);
    msetIntegers.insert (5);
    msetIntegers.insert (5);

    set <int> setIntegers;
    setIntegers.insert (5);
```

```

    setIntegers.insert (5);
    setIntegers.insert (5);

    cout << "Displaying the contents of the multiset: ";
    DisplayContent (msetIntegers);
    cout << endl;

    cout << "Displaying the contents of the set: ";
    DisplayContent (setIntegers);
    cout << endl;

    return 0;
}

```

## 第 21 章

### 测验

1. 默认排序标准由 `std::less` 指定。
2. 彼此相邻。
3. `size()`。
4. 在 `map` 中找不到重复的元素!

### 练习

1. 允许重复元素的关联容器，如 `std::multimap`：  
`std::multimap <string, string> multimapPeopleNamesToNumbers;`

2. 下面是一种解决方案：

```

struct fPredicate
{
    bool operator< (const wordProperty& lsh, const wordProperty& rsh) const
    {
        return (lsh.strWord < rsh.strWord);
    }
};

```

3. 参考第 20 章中练习 3 的解决方案。

## 第 22 章

### 测验

1. 一元谓词。
2. 它可以显示数据或计算元素个数。
3. 在 C++ 中，在应用程序运行阶段存在的所有实体都是对象，因此结构和类也可用作函数，这称为函数对象。注意，函数也可通过函数指针来调用，它们也是函数对象。

### 练习

1. 下面是一种解决方案：

```

template <typename elementType=int>
struct Double
{
    void operator () (const elementType element) const
    {
        cout << element * 2 << ' ';
    }
};

```

可以这样使用该一元谓词：

```
int main ()
{
    vector<int> vecIntegers;

    for (int nCount = 0; nCount < 10; ++ nCount)
        vecIntegers.push_back (nCount);

    cout << "Displaying the vector of integers: " << endl;

    // Display the array of integers
    for_each ( vecIntegers.begin ()           // Start of range
              , vecIntegers.end ()           // End of range
              , Double<> () ); // Unary function object

    return 0;
}
```

2. 添加一个整型成员，每次调用 operator()时都将递增该成员：

```
template <typename elementType=int>
struct Double
{
    int m_nUsageCount;

    // Constructor
    Double () : m_nUsageCount (0) {};

    void operator () (const elementType element) const
    {
        ++ m_nUsageCount;
        cout << element * 2 << ' ';
    }
};
```

3. 该二元谓词的定义如下：

```
template <typename elementType>
class CSortAscending
{
public:
    bool operator () (const elementType& num1,
                     const elementType& num2) const
    {
        return (num1 < num2);
    }
};
```

可以这样使用该谓词：

```
int main ()
{
    std::vector<int> vecIntegers;

    // Insert sample numbers: 100, 90... 20, 10
    for (int nSample = 10; nSample > 0; -- nSample)
        vecIntegers.push_back (nSample * 10);

    std::sort ( vecIntegers.begin (), vecIntegers.end (),
                CSortAscending<int> () );

    for ( size_t nElementIndex = 0;
          nElementIndex < vecIntegers.size ();
          ++ nElementIndex )
        cout << vecIntegers [nElementIndex] << ' ';

    return 0;
}
```

## 第 23 章

### 测验

1. 使用 `std::list::remove_if` 函数，因为它确保指向 `list` 中（未被删除的）元素的现有迭代器仍有效。
2. 如果没有显式指定谓词，`list::sort`（或 `std::sort`）将使用 `std::less<>`，这将使用运算符 `<` 对集合中

的对象进行排序。

3. 对指定范围内的每个元素调用一次。
4. `for_each` 返回函数对象。

### 练习

1. 下面是一种解决方案：

```
struct CaseInsensitiveCompare
{
    bool operator() (const string& str1, const string& str2) const
    {
        string str1Copy (str1), str2Copy (str2);

        transform (str1Copy.begin (),
                    str1Copy.end(), str1Copy.begin (), tolower);
        transform (str2Copy.begin (),
                    str2Copy.end(), str2Copy.begin (), tolower);

        return (str1Copy < str2Copy);
    }
};
```

2. 下面是一个演示程序。请注意 `std::copy` 是如何在不知道集合特征的情况下进行复制的。它只使用迭代器类。

```
#include <vector>
#include <algorithm>
#include <list>
#include <string>
#include <iostream>

using namespace std;

int main ()
{
    list <string> listNames;
    listNames.push_back ("Jack");
    listNames.push_back ("John");
    listNames.push_back ("Anna");
    listNames.push_back ("Skate");

    vector <string> vecNames (4);
    copy (listNames.begin (), listNames.end (), vecNames.begin ());

    vector <string> ::const_iterator iNames;
    for (iNames = vecNames.begin (); iNames != vecNames.end (); ++ iNames)
        cout << *iNames << ' ';

    return 0;
}
```

3. `std::sort` 与 `std::stable_sort` 之间的区别在于，后者在排序时保持对象的相对位置不变。由于该应用程序需要按生成顺序存储数据，因此应使用 `stable_sort`，以保持天体事件的相对顺序不变。

## 第 24 章

### 测验

1. 可以，通过提供一个谓词。
2. 运算符<。
3. 不能，只能操作栈顶元素。因此，不能访问栈底的 `CCoins` 对象。

### 练习

1. 该二元谓词可以是运算符<；



```

class CPerson
{
public:
    int m_nAge;
    bool m_bIsFemale;

    bool operator< (const CPerson& anotherPerson) const
    {
        bool bRet = false;
        if (m_nAge > anotherPerson.m_nAge)
            bRet = true;
        else if (m_bIsFemale && anotherPerson.m_bIsFemale)
            bRet = true;

        return bRet;
    }
};

```

2. 只需将字符串依次插入到栈中。弹出数据时，字符串的排列顺序便反转了，因为栈是一种后进先出容器。

## 第 25 章

### 测验

1. 不能。bitset 可存储的位数在编译阶段就已确定。
2. 因为它不能像其他容器那样动态地调整长度，它也不像容器那样支持迭代器。
3. 不会。在这种情况下使用 std::bitset 最合适。

### 练习

1. 在这种情况下使用 std::bitset 最合适。

```

#include <bitset>
#include <iostream>

int main()
{
    // Initialize the bitset to 1001
    std::bitset<4> fourBits (9);

    std::cout << "fourBits: " << fourBits << std::endl;

    // Initialize another bitset to 0010
    std::bitset<4> fourMoreBits (2);

    std::cout << "fourMoreBits: " << fourMoreBits << std::endl;

    std::bitset<4> addResult(fourBits.to_ulong() +
    ➤ fourMoreBits.to_ulong());
    std::cout << "The result of the addition is: " << addResult;

    return 0;
}

```

2. 对前一个示例中的 bitset 对象调用 flip 函数：

```

addResult.flip ();
std::cout << "The result of the flip is: " << addResult << std::endl;

```

## 第 26 章

### 测验

1. 我会先看看 [www.boost.org](http://www.boost.org)，希望您也如此！
2. 不会。一般而言，如果智能指针编写得好（且选择正确）的话是不会的。

3. 如果是侵入式的, 将由指针拥有的对象保存引用计数; 否则, 指针可将这种信息保存在自由存储区中的共享对象中。

4. 需要双向遍历链表, 因此它必须是双向链表。

### 练习

1. 语句 `pObject->DoSomething ();` 有问题, 因为指针在复制时失去了对对象的拥有权。这将导致程序崩溃 (或发生令人非常不愉快的事情)。

2. 不会出现切片问题, 因为 `auto_ptr` 将转交对象的所有权而不是复制它。

## 第 27 章

### 测验

1. 插入运算符 (`<<`) 是 `ostream` 对象的一个成员运算符, 用于写入输出设备。

2. 提取运算符 (`>>`) 是 `istream` 对象的一个成员运算符, 用于写入程序的变量。

3. `get()` 的第一种形式没有参数, 它返回找到的字符值, 如果到达文件尾则返回 EOF。

`get()` 的第二种形式接受一个字符引用参数, 将输入流中的下一个字符赋给该字符变量, 并返回一个 `istream` 对象。

第三种形式的 `get()` 接受一个数组、最多要读取的字符数和终止字符作为参数, 除非遇到终止字符, 否则它将比最大字符数参数少一个字符存储到数组中 (在末尾加上空字符), 如果遇到终止字符, 则在数组末尾加上空字符, 并将终止字符留在输入流中。

4. `cin.read()` 用于读取二进制数据结构。

`getline()` 用于从 `istream` 的缓冲区中读取。

5. 宽度刚好显示整个数。

6. `istream` 对象引用。

7. 要打开的文件名称。

8. `ios::ate` 跳到文件末尾, 但用户可以在文件的任何地方写入数据。

### 练习

1. 下面是一种解决方案:

```
#include <iostream>
int main()
{
    int x;
    std::cout << "Enter a number: ";
    std::cin >> x;
    std::cout << "You entered: " << x << std::endl;
    std::cerr << "Uh oh, this to cerr!" << std::endl;
    std::clog << "Uh oh, this to clog!" << std::endl;
    return 0;
}
```

2. 下面是一种解决方案:

```
#include <iostream>
int main()
{
    char name[80];
    std::cout << "Enter your full name: ";
    std::cin.getline(name, 80);
    std::cout << "\nYou entered: " << name << std::endl;
    return 0;
}
```



## 3. 下面是一种解决方案:

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    cout << "enter a phrase: ";
    while ( cin.get(ch) )
    {
        switch (ch)
        {
            case '!':
                cout << '$';
                break;
            case '#':
                break;
            default:
                cout << ch;
                break;
        }
    }
    return 0;
}
```

## 4. 下面是一种解决方案:

```
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char**argv)    // returns 1 on error
{
    if (argc != 2)
    {
        cout << "Usage: argv[0] <infile>\n";
        return(1);
    }

    // open the input stream
    ifstream fin (argv[1],ios::binary);
    if (!fin)
    {
        cout << "Unable to open " << argv[1] << " for reading.\n";
        return(1);
    }

    char ch;
    while ( fin.get(ch))
        if ((ch > 32 && ch < 127) || ch == '\n' || ch == '\t')
            cout << ch;
    fin.close();
}
```

## 5. 下面是一种解决方案:

```
#include <iostream>

int main(int argc, char**argv)    // returns 1 on error
{
    for (int ctr = argc-1; ctr>0 ; ctr--)
        std::cout << argv[ctr] << " ";
}
```

## 第 28 章

## 测验

1. 异常是使用关键字 throw 创建的对象,用于指示异常状态,沿调用栈向上传递到能够处理它的第一条 catch 语句。
2. try 块是一组可能导致异常的语句。

3. catch 语句是一个例程，其参数指出了它能处理的异常类型。它位于 try 块后面，用于捕获 try 块内可能引发的异常。

4. 异常是一个对象，可包含在用户创建的类中能定义的任何信息。

5. 程序调用关键字 throw 时将创建异常对象。

6. 通常应该按引用来传递。如果不打算修改异常对象的内容，应传递 const 引用。

7. 如果按引用传递异常就可以。

8. catch 语句按它们出现在源代码中的顺序被检查，将使用特征标与异常匹配的第一条 catch 语句。一般而言，最好按具体到通用的顺序捕获异常。

9. catch(...)捕获任何类型的异常。

10. 断点是代码中调试器停止执行的地方。

## 练习

1. 下面是一种答案：

```
#include <iostream>
using namespace std;
class OutOfMemory {};
int main()
{
    try
    {
        int *myInt = new int;
        if (myInt == 0)
            throw OutOfMemory();
    }
    catch (OutOfMemory)
    {
        cout << "Unable to allocate memory!" << endl;
    }
    return 0;
}
```

2. 下面是一种答案：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
class OutOfMemory
{
public:
    OutOfMemory(char *);
    char* GetString() { return itsString; }
private:
    char* itsString;
};

OutOfMemory::OutOfMemory(char * theType)
{
    itsString = new char[80];
    char warning[] = "Out Of Memory! Can't allocate room for: ";
    strncpy(itsString,warning,60);
    strncat(itsString,theType,19);
}

int main()
{
    try
    {
        int *myInt = new int;
        if (myInt == 0)
            throw OutOfMemory("int");
    }
    catch (OutOfMemory& theException)
    {
        cout << theException.GetString();
    }
}
```

资源解密网  
PDG

```

    }
    return 0;
}

```

### 3. 下面是一种答案:

```

#include <iostream>
using namespace std;
// Abstract exception data type
class Exception
{
public:
    Exception(){}
    virtual ~Exception(){}
    virtual void PrintError() = 0;
};

// Derived class to handle memory problems.
// Note no allocation of memory in this class!
class OutOfMemory : public Exception
{
public:
    OutOfMemory(){}
    ~OutOfMemory(){}
    virtual void PrintError();
private:
};

void OutOfMemory::PrintError()
{
    cout << "Out of Memory!!" << endl;
}

// Derived class to handle bad numbers
class RangeError : public Exception
{
public:
    RangeError(unsigned long number){badNumber = number;}
    ~RangeError(){}
    virtual void PrintError();
    virtual unsigned long GetNumber() { return badNumber; }
    virtual void SetNumber(unsigned long number) {badNumber = number;}
private:
    unsigned long badNumber;
};

void RangeError::PrintError()
{
    cout << "Number out of range. You used " ;
    cout << GetNumber() << "!!" << endl;
}

void MyFunction(); // func. prototype
int main()
{
    try
    {
        MyFunction();
    }
    // Only one catch required, use virtual functions to do the
    // right thing.
    catch (Exception& theException)
    {
        theException.PrintError();
    }
    return 0;
}

void MyFunction()
{
    unsigned int *myInt = new unsigned int;
    long testNumber;

    if (myInt == 0)

```

```

        throw OutOfMemory();

    cout << "Enter an int: ";
    cin >> testNumber;

    // this weird test should be replaced by a series
    // of tests to complain about bad user input

    if (testNumber > 3768 || testNumber < 0)
        throw RangeError(testNumber);

    *myInt = testNumber;
    cout << "Ok. myInt: " << *myInt;
    delete myInt;
}

```

#### 4. 下面是一种答案:

```

#include <iostream>
using namespace std;
// Abstract exception data type
class Exception
{
public:
    Exception(){}
    virtual ~Exception(){}
    virtual void PrintError() = 0;
};
// Derived class to handle memory problems.
// Note no allocation of memory in this class!
class OutOfMemory : public Exception
{
public:
    OutOfMemory(){}
    ~OutOfMemory(){}
    virtual void PrintError();
private:
};

void OutOfMemory::PrintError()
{
    cout << "Out of Memory!!\n";
}

// Derived class to handle bad numbers
class RangeError : public Exception
{
public:
    RangeError(unsigned long number){badNumber = number;}
    ~RangeError(){}
    virtual void PrintError();
    virtual unsigned long GetNumber() { return badNumber; }
    virtual void SetNumber(unsigned long number) {badNumber = number;}
private:
    unsigned long badNumber;
};

void RangeError::PrintError()
{
    cout << "Number out of range. You used ";
    cout << GetNumber() << "!!" << endl;
}

// func. prototypes
void MyFunction();
unsigned int * FunctionTwo();
void FunctionThree(unsigned int *);

int main()
{
    try
    {
        MyFunction();
    }
}

```



```

    // Only one catch required, use virtual functions to do the
    // right thing.
    catch (Exception& theException)
    {
        theException.PrintError();
    }
    return 0;
}

unsigned int * FunctionTwo()
{
    unsigned int *myInt = new unsigned int;
    if (myInt == 0)
        throw OutOfMemory();
    return myInt;
}

void MyFunction()
{
    unsigned int *myInt = FunctionTwo();
    FunctionThree(myInt);
    cout << "Ok. myInt: " << *myInt;
    delete myInt;
}

void FunctionThree(unsigned int *ptr)
{
    long testNumber;
    cout << "Enter an int: ";
    cin >> testNumber;
    // this weird test should be replaced by a series
    // of tests to complain about bad user input
    if (testNumber > 3768 || testNumber < 0)
        throw RangeError(testNumber);
    *ptr = testNumber;
}

```

5. 在处理“内存耗尽”状态的过程中，xOutOfMemory 的构造函数创建了一个 string 对象。这种异常仅在程序耗尽内存时才会出现，因此该内存分配操作将失败。

试图创建该 string 对象可能引发相同的异常，这将形成无限循环，直到程序崩溃。如果确实需要该 string 对象，可在程序开始前在静态缓冲区中分配空间，然后在该异常被引发时使用它。要测试该程序，可将 if (var == 0) 修改为 if (1)，这将引发内存耗尽异常。

## 第 29 章

### 测验

1. 用于在调试阶段检查条件。断言是一种验证变量和条件（如字符串指针是否有效）的强大方式，同时不会降低应用程序的运行性能。
2. 将其用于跟踪和记录，以获悉哪个文件包含要记录的事件。
3. 两个字节为 16 位，因此最多可存储 16 位值。
4. 5 位能够存储 32 个不同的值（0~31）。
5. 结果为 1111 1111。
6. 结果为 0011 1100。

### 练习

1. 防范多重包含头文件 STRING.H 的语句如下：

```

#ifndef STRING_H
#define STRING_H
...
#endif

```

## 2. 下面是一种答案:

```
#include <iostream>

using namespace std;
#ifndef DEBUG
#define ASSERT(x)
#elif DEBUG == 1
#define ASSERT(x) \
    if (! (x)) \
    { \
        cout << "ERROR!! Assert " << #x << " failed" << endl; \
    }
#elif DEBUG == 2
#define ASSERT(x) \
    if (! (x) ) \
    { \
        cout << "ERROR!! Assert " << #x << " failed" << endl; \
        cout << " on line " << __LINE__ << endl; \
        cout << " in file " << __FILE__ << endl; \
    }
#endif
```

## 3. 下面是一种答案:

```
#ifndef DEBUG
#define DPRINT(string)
#else
#define DPRINT(string) cout << #string ;
#endif
```

## 4. 下面是一种答案:

```
class myDate
{
public:
    // stuff here...
private:
    unsigned int Month : 4;
    unsigned int Day : 8;
    unsigned int Year : 12;
}
```

“这是一本真正适合C++程序设计初学者和没有任何编程经验的人的优秀图书。”

—— 独立评论人

## 21天学通C++ (第6版)

只需每天一小时便可具备开始使用C++进行编程所需的全部技能。通过阅读这本内容全面的教程，读者可快速掌握基本知识并学习更高级的特性和概念：

掌握有关C++和面向对象编程的基本知识；

学习一些C++高级特性；

学习标准模板库以及大多数真实C++应用程序都用到了的容器和算法；

向在公司环境中实现C++的权威人士学习专家级技巧。

自己掌握学习时间和学习步伐：

- 不需要任何编程经验；
- 学习C++以及面向对象设计、编程和分析；
- 编写快速而功能强大的C++程序、编译源代码以及创建可执行文件；
- 了解最新的ANSI标准；
- 使用标准模板库中的算法和容器编写功能丰富而稳定的C++应用程序；
- 使用函数、数组、变量和智能指针完成复杂的编程工作；
- 学习使用继承和多态扩展程序的功能；
- 通过向编程专家学习，掌握C++特性；
- 适用于任何ANSI C++编译器。

**SAMS**

封面设计：胡萍丽

分类建议：计算机 / 程序设计 / C++

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-20793-7



9 787115 207937 >

ISBN 978-7-115-20793-7 /TP

定价：55.00 元